Quick Update: Delphi - Utilidades e Novidades

Clube Delphi

Ano 9 :: Edição 130 :: R$ 11,90

DevMedia

dbExpress
Transações

DataSnap XE
Técnicas avançadas

Boas Práticas
Padrões Criacionais

Acesso ao BD no Prism

Usando ADO.NET Entity Framework

Delphi Prism
Entendendo o que são Assemblies, Class Libraries e GAC

Delphi Prism
Conheça os melhores recursos da linguagem – Parte 4

ISSN 1517990-7

Dificuldades em encontrar conteúdo sobre .Net para iniciantes?

A DevMedia simplifica pra você!

easy .net magazine

Easy .Net Magazine:

A primeira revista digital feita especialmente para quem está dando os primeiros passos em .Net!

Você encontrará exemplos e modelos fáceis e práticos, noções introdutórias e conceitos básicos.

Conheça as vantagens:

- Vídeos e apresentações "embutidos" nas páginas;
- Ilustrações, diagramas e modelos apresentados em PowerPoint;
- Entrevistas em vídeo;
- Códigos-fonte que podem ser copiados com CTRL+C/CTRL+V.

A Easy.Net não é feita em PDF ou Flash!
Ela utiliza um aplicativo 100% online.

DevMedia group

Acesse agora a edição Nº 1 totalmente grátis e comprove

www.devmedia.com.br/easynet

Easy.Net Magazine, seu curso inicial de .Net!

Sumário

Delphi

Novidades – Mini-Curso

06 – Delphi Prism
Melhorias da linguagem – Parte 4
[*Rodrigo Araujo*]

.Net

13 – Delphi Prism
Entendendo o que são Assemblies, Class Libraries e GAC
[*Guinther Pauli*]

Banco de Dados

20 – dbExpress
Conceitos relacionados a transações em servidores de BD relacionais
[*Jéfferson Ricardo Zuchi*]

Boas Práticas

Boas Práticas

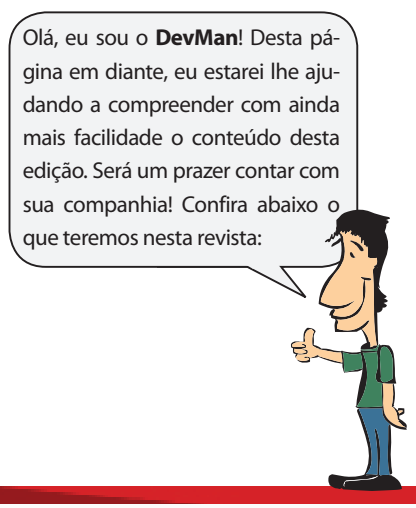
26 – DataSnap XE
Técnicas avançadas
[*Alexandre José Gonçalves da Silva*]

Boas Práticas

34 – Padrões Criacionais
Fabrique famílias de objetos
[*Rafael Stavarengo*]

Boas Práticas

43 – Acesso a dados no Delphi Prism
Mapeamento Objeto / Relacional e Persistência com ADO.NET Entity Framework
[*Rodrigo Araujo*]



Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.CD** Senha: **MXJ132**

Clube Delphi

Ano 9 - 130ª Edição 2011 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Leandro Daniel
contato@leandrodaniel.com

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Jornalista Responsável
Kaline Dolabella - JP24185

Capa e Diagramação
Romulo Araujo - romulo@devmedia.com.br

Coordenação Geral e Revisão
Daniella Costa - daniella@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor
A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Uellen Goulart – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade
Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Cristiany Queiróz

publicidade@devmedia.com.br

EDITORIAL

Esta edição destaca o acesso a dados no Delphi Prism. Só para se ter uma ideia, existem dezenas de formas de acesso a dados no .NET Framework, motivo que causa muita confusão, principalmente para quem está iniciando na plataforma. Entre as formas de acesso, podemos citar o uso de ADO.NET através de classes como SqlConnection e SqlCommand (no caso do SQL Server), ou ainda usar o ADO.NET em uma camada separada, como um Data Access Layer, usando boas práticas e separação de responsabilidades, isolando o domínio. DataSets tipados e TableAdapters também são uma boa forma de acesso a dados, pois permitem separar o acesso em um layer independente da UI. Usando o ASP.NET temos o modelo baseado em DataSource, que tira proveito do mecanismo de DataBinding, incluindo o SqlDataSource e ObjectDataSource. Para um desenvolvimento mais profissional e orientado a objetos, podemos usar um framework como o NHibernate ou o ADO.NET Entity Framework.

O ADO.NET Entity Framework (EF) é a ferramenta de persistência e mapeamento objeto / relacional nativa do .NET Framework, concorrente direto do NHibernate. Ele originalmente apareceu pela primeira vez no Service Pack 1 do Visual Studio 2008 / .NET 3.5. No Visual Studio 2010 / .NET 4.0 ele está na sua versão 2, apesar de normalmente nos referenciarmos a ele por EF4 (versão 4). O ADO.NET EF permite o desenvolvimento de aplicações para .NET onde o foco principal é a orientação a objetos, sem uma comunicação direta com um servidor de banco de dados relacional, como o SQL Server. É o framework que gera todos os comandos SQL. A última versão permite que o desenvolvedor inicie a criação de um aplicativo pela definição do domínio (entidades), com um diagrama de classes, podendo a seguir gerar todo o esquema para o SGBDR. Essa técnica é conhecida como Model First Development. Se já possui um banco existente, o EF pode fazer a engenharia reversa, gerando entidades no modelo a partir do BD. Outra característica interessante do EF é a capacidade de suportar tipos complexos, uma novidade da última versão, bem como Lazy Loading, que permite o carregamento de entidades relacionadas conforme utilização. O que considero um dos melhores recursos é o novo suporte a POCO (Plain Old CLR Objects), onde podemos modelar o domínio da aplicação, com as entidades, sem gerar código que esteja atrelado ao framework ou a uma tecnologia em específico. As classes do domínio simplesmente mapeiam a tabela do banco, normalmente sem conter métodos ou derivar de uma classe base para que possam funcionar. Isso dá uma maior flexibilidade e independência ao se trabalhar com o EF. Isso é importante para aplicações em camadas, que prezam por baixo acoplamento entre suas partes, seguindo bons princípios do DDD (Domain Driven Design). Nesta edição, damos especial destaque ao uso dessa poderosa ferramenta para acesso a dados, utilizando o Delphi Prism XE.

Guinther Pauli
guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Linux
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET,C#)
guintherpauli.blogspot.com - twitter.com/guintherpauli

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia

Sistema de créditos DevMedia

Conteúdo **completo** ao seu alcance

Com o sistema de créditos DevMedia você pode comprar individualmente mais de 4.000 vídeo aulas, 8000 artigos, além de palestras e vídeos com formações completas.

TODAS as vídeo aulas que você sempre quis ver a partir de

R$ 0,99

ECONOMIA: você não precisa assinar 8 produtos diferentes para ter acesso ao conteúdo completo do site!

Saiba mais sobre o Sistema de Créditos!

www.devmedia.com.br/creditos

DevMedia

Delphi Prism
Melhorias da linguagem – Par

A palavra delegate é utilizada para definir tipos de eventos personalizados, com ela é possível criar um novo tipo de evento customizado que poderá ser utilizado em uma determinada classe. A definição de delegate é algo semelhante ao conceito de objetos para procedures que está presente em outras versões anteriores do Delphi.

O tipo delegate nos permite encapsular a referência de um método dentro de um objeto e este mesmo objeto pode ser transferido entre chamadas através das classes de uma aplicação para ser invocado em algum outro ponto do programa, desta forma não se sabendo em tempo de compilação qual método será invocado. Simplificando esta ideia inicial, por exemplo, pode-se encapsular um método dentro de um objeto e passar este objeto como argumento entre chamadas de métodos e em algum determinado ponto do código resgatar esta referência e invocar o método pelo qual o delegate foi inscrito. Assim o comportamento de execução do método será realizado de acordo com sua referência assinada.

Delegates possuem estreita ligação com a estrutura de métodos e é declarado com as mesmas características de um método, possuem valor de retorno e parâmetros. Uma importante observação é que para escrever um método através de um delegate, a declaração do método e do delegate devem ser compatíveis. Se um método tem como retorno um tipo inteiro e dois parâmetros de entrada do tipo string o delegate que irá representá-lo em uma chamada anônima deverá possuir uma estrutura idêntica, caso contrário o código fonte apresentará erros.

O conceito de delegate se originou na ideia de função para ponteiros que é bastante difundida dentro do universo de programação como linguagens C/C++. Uma relevante e considerável melhoria que o .NET trouxe em relação aos ponteiros para função do C/C++ não gerenciado é a vantagem dos delegates serem tipos seguros, pois as funções para ponteiros são representadas apenas pelo endereço em memória e não dizem nada a respeito da quantidade de parâmetros ou tipos e valores de retorno. Com isso, se em tempo de desenvolvimento o programador se equivocar na declaração destas estruturas, este erro somente será

Resumo DevMan

De que trata o artigo:

O artigo irá tratar de demonstrar algumas das melhorias e novidades na utilização da linguagem do Delphi Prism destacando com trechos de códigos as funcionalidades presentes nesta versão que não se enquadram em versões anteriores do Delphi.

Para que serve:

Os recursos demonstrados neste artigo servem para facilitar e acelerar a codificação, incluindo novas funcionalidades avançadas presentes na linguagem, além de aprimorar, aperfeiçoar e realçar o que cada recurso apresenta de melhor para o desenvolvimento ajudando inclusive na orientação a objetos.

Em que situação o tema é útil:

O tema é útil para auxiliar aqueles que já possuem conhecimento na linguagem Delphi e que gostariam de verificar o que há de novo no Delphi Prism, dessa forma, o artigo auxiliará na ampliação dos conhecimentos focados diretamente em pontos chave para que tornem o desenvolvimento de código mais enxuto, simplificado e com qualidade.

Delphi Prism – Melhorias da linguagem :

No desenvolvimento do artigo será visto o uso de Delegates para encapsular a referência de um método ou eventos personalizados. Logo após, o uso de "Type" para indicar a definição de um ou mais tipos no Delphi Prism. Após, surge "Abstract" para a definição de membros de uma classe, permitindo que a classe onde ocorre a definição não possua a implementação desses membros. O uso de "Final" para indicar que o código a ser definido é final e não poderá ser modificado por classes descendentes. A utilização de "Has" para definições de contextos genéricos. E, por dar continuidade a contextos genéricos, surge "Is" possibilitando indicar um determinado tipo específico para uso. Além disso, aparece o "Iterator" realizando uma codificação de forma mais simplificada de sequências customizadas de dados. A palavra "Of" para realizar referência a determinadas classes. A grande funcionalidade do uso de "Default" para definições de propriedades relacionadas a matrizes e classes. "Inline" determinando estruturas de tamanho fixo incorporadas em blocos de códigos. As restrições para somente leitura através de "readonly". A codificação utilizando "add" e "remove" em eventos. E, por final, a utilização de códigos "unsafe" e "pinned" possibilitando o uso de recursos de maneira não gerenciada.

te 4

Rodrigo Araujo

descoberto em tempo de execução e não tem como prever o comportamento da execução deste código. Já não é assim com os delegates, pois possuem uma estrutura de metadados bem definida que descreve seus atributos, não acontecendo o cenário citado anteriormente. A **Listagem 1** indica um exemplo do uso de delegate para definição de um tipo personalizado.

Type

A utilização de Type é indicada para iniciar a definição de um ou mais tipos no Delphi Prism. Ao realizar uso desta palavra, o Prism entenderá que irá existir a declaração de um novo tipo ao escrever o seu nome e utilizar definições já existentes seja pelo nome de tipo escrito por completo ou apenas ao namespace atual presente no código. O uso de Type poderá estar relacionado, principalmente, a classes, interfaces, records, enums, sets, tipos genéricos, arrays, tipos anônimos, delegates entre outros.

A declaração é iniciada pela palavra type seguida do novo nome que estará relacionado a um tipo. Este tipo poderá utilizar os que estão presentes no código fonte que está sendo informado ou então utilizar as definições existentes em cada namespace, onde será utilizado o nome do namespace seguido do tipo a ser utilizado. A **Listagem 2** demonstra o uso de type para a definição de dois tipos, exemplificando as duas situações mencionadas. Nesta listagem, o primeiro exemplo, de maneira bastante simples, define uma classe pública chamada de ClasseModelo e que apresentará um atributo do tipo string chamado de Teste. Já o segundo exemplo, cria uma Lista<T> e utilizará a definição presente em System.Collections.ICollection.

Abstract

Os membros de uma classe poderão ser definidos como abstratos, dessa forma, possibilitando que a classe onde está sendo realizada a definição não possua a definição de código desses membros, mas apenas indique a estrutura para utilização em classes descendentes.

Abstract permite definir onde realmente devem ser codificados os métodos em uma hierarquia. O seu uso implica que determinado método não tenha implementação na classe onde foi declarado. A definição de código de fato deve ocorrer nos seus descendentes.

Ao definir pelo menos um método como abstrato, determina-se que a classe normalmente não vai ser instanciada e também será abstrata. Somente métodos virtuais ou dinâmicos podem ser abstratos.

O uso de Abstract facilita a programação e definição de classes descendentes, assim permitindo que cada uma realize de maneira diferenciada e pontual cada uma das implementações das funcionalidades desejadas. A **Listagem 3** possibilita visualizar melhor o tratamento de Abstract na definição de um membro de uma classe e o seu uso por uma classe descendente que irá realizar a implementação do código.

Listagem 1. Código para demonstrar o uso de delegate

```
type
   MeuEvento = delegate (Sender: Object; Args: EventArgs);
```

Listagem 2. Uso de type para a definição de tipos

```
type
  ClasseModelo = public class
  public
    property Teste: String;
  end;
type
  Lista<T> = public class(Arraylist, System.Collections.ICollection)
  end;
```

Listagem 3. Definição de um membro abstrato em uma classe

```
type

  Pessoa = public class
  public
    property Nome: String;
    procedure GeraSenha(Nome : string); virtual; abstract;
  end;

type
  Aluno = public class(Pessoa)
  public
    procedure GeraSenha(Nome : string); override;
  end;

implementation

procedure Aluno.GeraSenha(Nome: String);
begin
  //definição de código para geração de senha do aluno
end;
```

O código da **Listagem 3** permite observar que a classe aluno herda as características da classe pessoa, dessa forma, obviamente, um aluno é uma pessoa, e o que irá diferenciar a classe aluno é a implementação de código para a procedure GeraSenha que possuirá a definição de código de acordo com as regras de geração de senha estabelecidas para um aluno. Tal característica poderá ser utilizada para outros tipos de pessoas, como professores, funcionários, visitantes entre outros.

Nota

Observe que o uso de override, garante que apenas uma implementação do método exista em tempo de execução nas classes descendentes, ocultando as implementações dos ancestrais. Porém, esses métodos devem ter exatamente a mesma assinatura, ou seja, mesmo nome do método e mesmo número, ordem e tipos dos parâmetros.

Nota

A palavra reservada Final é utilizada nos métodos virtuais ou dinâmicos de uma classe, quando se deseja impedir que esses métodos sejam sobrepostos em suas classes herdadas. Se você tentar criar uma classe descendente que sobreponha o método, utilizando um override, receberá uma mensagem de erro.

Listagem 4. Uso de Final em uma classe

```
type
 apartamento = class (Object)
 public
  method x(); virtual; final;
 end;
```

Listagem 5. Código utilizando Has

```
type
  Lista<T> = public class(Arraylist, System.Collections.ICollection)
    Where T is Object, T has constructor;
  end;
```

Listagem 6. Código de uma estrutura condicional utilizando Is

```
Method MainForm.TesteString(palavra : Object);
begin
  if palavra is String then
  begin
   MessageBox.Show("A string contém "+ palavra.ToString().Length.
   ToString()+" caracteres.");
  end;
end;
```

Assim, a definição de GeraSenha poderá ser estabelecida de acordo com as regras para cada tipo de pessoa levando em conta a utilização de Abstract na definição do código a ser criado. Neste caso, esta é uma boa prática para o uso de Abstract na programação de classes que se diferenciam em alguns aspectos, mas com propriedades iniciais semelhantes.

Neste aspecto levando em conta as definições e características de polimorfismo podem ser citadas para melhor exemplificar as situações presentes na definição de classes e uso de abstract.

O polimorfismo representa uma característica da Orientação a Objetos que permite aos objetos das classes terem comportamentos diferentes, sejam de seus ancestrais, da própria classe ou mesmo de outras classes. Assim, os métodos definidos e implementados em classes ancestrais, podem ser redefinidos e reimplementados em classes descendentes.

Para que isso funcione adequadamente, é necessário garantir qual a implementação de um método será executada, visto que podem existir mais de uma com o mesmo nome, seja na hierarquia ou dentro da própria classe.

Final

A utilização de Final em construções de código irá indicar que o código a ser definido é final e não poderá ser substituído por classes descendentes. Seu uso é aplicado a códigos que possuam métodos virtuais ou override. Na **Listagem 4** é visualizada a sua utilização em uma classe, esta classe é chamada de apartamento e possui um método X que irá ser descrito contendo a declaração Final na sua definição, dessa forma ao criar o código para o método, este código será único e não serão permitidas alterações em alguma classe descendente.

Has

A palavra reservada Has é utilizada em definições de contextos de tipos genéricos, onde será apresentada em conjunto com cláusulas where, assim, neste aspecto, a cláusula where é observada para aplicar restrições ou filtros a construtores com o objetivo de que classes descendentes levem em conta as regras pré-definidas de maneira adequada. A **Listagem 5** indica a utilização de Has na definição de uma lista para objetos.

O código da **Listagem 5** determina a criação de uma lista e a demonstração do uso de Has na criação desta lista e ela terá restrições que são definidas através da cláusula where indicando que T deverá ser um Object para poder representar um item desta lista.

Além disso, a **Listagem** 5 representa uma coleção que indica determinado tipo de classes especializadas para armazenamento e recuperação de dados. São implementadas através de interfaces, portanto, se o programador quiser pode criar sua própria coleção, basta implementar as interfaces disponibilizadas no .NET Framework. As classes podem ser encontradas, principalmente, nos namespaces System.Collections, System.Collections.Generics, System.Collections.Specialized e System.Collections.ObjectModel. Dentre as várias coleções disponíveis no .NET Framework, além da própria e clássica coleção representada por ArrayList.

Is

Esta palavra chave, em Delphi Prism apresenta dois contextos principais. O primeiro, e já tradicional, é o emprego em estruturas condicionais If representados na **Listagem 6**. O segundo é de uso semelhante ao descrito por Has em contextos de tipos genéricos permitindo que seja indicado um determinado tipo específico de uso. Além disso, utilizando Generics o Delphi Prism também suporta definições como "is record" e "is class". A **Listagem 7** demonstra o uso de Is indicando a construção de uma classe.

A **Listagem 6** realiza o uso de Is para verificação de tipos onde poderão ser

realizadas determinadas operações para tipos distintos de dados. Neste caso, foi realizado um tratamento diferenciado para o tipo String, um tratamento semelhante poderia ser usado para outros tipos de dados.

Na **Listagem 7** foi definida uma classe chamada de TipoNovo onde uma restrição indicada no where é de que T deverá ser do tipo record, com isto delimitando os dados somente a este tipo estabelecido.

Iterator

Um Iterator possibilita codificar de maneira simples e fácil sequências customizadas de dados. São tipos especiais de métodos que retornam determinadas sequências que, ao ser executado o método, um novo objeto deste método é retornado e é este que irá representar a sequência definida. Em conjunto com o Iterator poderá trabalhar o Yield, uma vez que o código começará a enumerar a sequência de saída a cada iteração da estrutura utilizada e o Iterator será executado para fornecer os elementos desta sequência, onde o Yield será utilizado para sair da estrutura armazenando o seu estado no momento da saída.

O código da **Listagem 8** demonstra a utilização de um Iterator em conjunto com um Yield para retornar uma sequência de inteiros que irão representar os números pares de zero a duzentos.

As duas primeiras linhas indicam a definição do método como sendo uma sequência de inteiros e a utilização do Iterator, dessa forma, o compilador através da diretiva de Iterator saberá que o método não é comum e receberá um tratamento diferenciado devido à forma de iteração que será utilizada com Yield. Em seguida, ocorre a implementação do método, utilizando uma estrutura de repetição e a palavra Yield. A cada chamada de Yield um valor será armazenado na sequência e retornado para o método GetNumPares. Após armazenar o valor, a estrutura de repetição seguirá a sua execução indo para o próximo passo de execução, e assim por diante, até chegar ao seu término.

Para utilizar os resultados obtidos com a **Listagem 8**, o código da **Listagem 9** indica como verificar os dados armazenados no método. No código é utilizado o método GetNumPares e seus resultados serão exibidos através de mensagens na tela. A cada iteração da estrutura for each é apresentada uma mensagem com um valor armazenado na sequência de inteiros.

Of

Permite realizar a referência a determinadas classes ou metaclasses. Além disso, o Delphi Prism permite que seja executada a referência de classes de uma forma que seja compatível com outras linguagens relacionadas a .NET e seu suporte atende a classes com métodos virtuais, construtores virtuais, determinados tipos de classes e a utilização do operador new na construção de variáveis relacionadas às classes para realizar a sua chamada ao construtor. O código da **Listagem 10** demonstra o uso de Of para a criação de uma classe.

Na **Listagem 10** é definida uma classe chamada de ClasseTeste contendo um método virtual nomeado de Teste e uma referência é adicionada através de MinhaClasse para a ClasseTeste utilizando a palavra Of.

Listagem 7. Código de Is na construção de uma classe

```
type
 TipoNovo<T> = class
 Where T is record;
  FVal: T;
 public
  constructor (Val: T);
 end;
```

Listagem 8. Código utilizando Iterator

```
private
  method GetNumPares: sequence of Integer; iterator;

method MainForm.GetNumPares: sequence of Integer;
begin
  for i: Integer :=0 to 200 step 2 do
    yield i;
end;
```

Listagem 9. Utilizando o método definido por Iterator

```
Var numeros := GetNumPares();
for each n in numeros do
 MessageBox.Show(n.ToString());
```

Listagem 10. Utilização de Of através de uma classe

```
type
 ClasseTeste = public class
 public
  constructor; virtual;
  class method Teste; virtual;
 end;
 MinhaClasse = class of ClasseTeste;
```

Listagem 11. O uso da palavra Default

```
type
 Produtos = class
 Method BuscaItens(i: Integer): Integer;
 Property Itens[i: Integer]: Integer read BuscaItens; default;
end;
```

Listagem 12. Uso da classe Produtos

```
var teste : Produtos;
teste.Itens[5];
teste[5];
```

Nota

Em orientação a objetos, uma metaclasse é uma classe cujas instâncias também são classes e não objetos no sentido tradicional. Assim como classes definem o comportamento de certos objetos, metaclasses definem o comportamento de certas classes e suas instâncias.

Default

O uso de Default possibilita criar definições de propriedades para índices de matrizes permitindo que funcione como uma espécie de indexador para um determinado atributo de uma classe. Dessa forma o compilador saberá utilizar implicitamente este índice no momento em que uma instância da classe for utilizada. A **Listagem 11** demonstra o uso de Default na criação de uma classe.

Na **Listagem 11** foi definida uma classe chamada de Produtos esta classe possui

um método chamado BuscaItens e uma propriedade Itens, esta propriedade poderá ser acessada de forma implícita como a **Listagem 12** demonstra.

Através da **Listagem 12** é possível perceber as duas formas, a convencional e a implícita, onde se não existisse o uso de Default na definição da classe, não seria possível realizar a chamada a teste[5], pois ocorreria um erro em tempo de compilação.

Além disso, múltiplas definições de Default também são suportadas, realizando uma espécie de sobrecarga implícita para diferentes tipos de parâmetros. A **Listagem 13** demonstra o uso desta sobrecarga.

Dessa forma, de acordo com a **Listagem 13**, as seguintes possibilidades estão presentes na **Listagem 14**, que define algumas formas de utilizar a classe definida.

Pela **Listagem 14** é observada a sobrecarga e as possibilidades e facilidades que ela permite para o desenvolvimento. A herança permite que de forma fácil seja alterada a implementação de uma funcionalidade. Isso é obtido através da técnica de sobrecarga de operações. Qualquer alteração na classe base implica em alterações nas subclasses, o que não é recomendado quando se deseja que exista um modelo mais flexível.

Nota

Thread safe refere-se a aplicações que apresentam programação voltada a multithreads e com isso possibilitam execução de mais de uma thread ao mesmo tempo.

Nota

Deverá ser bem analisado o uso de readonly em uma classe, pois ao adicionar a palavra readonly, isto faz com que todos os campos e propriedades dela sejam de somente leitura e poderão ser apenas definidas pelo seu construtor, auxiliando na criação de classes imutáveis.

Inline

Possibilita declarar estruturas de tamanho fixo contendo matrizes incorporadas em blocos de códigos, onde estas matrizes somente poderão ser definidas dentro de uma classe ou de um registro. A **Listagem 15** indica o uso de inline na definição de um array de caracteres e um array de inteiros.

Pela **Listagem 15** foi definido um record de nome meuregistro, da mesma forma que poderia ter sido definida uma classe e, este record contém um array de caracteres e um array de inteiros para serem utilizados com inline.

De acordo com a definição, seus principais subtipos para uso em arrays são smallint, shortint, integer, int64, byte, word, cardinal, uint64, double, single, boolean e char.

Readonly

É utilizado para criar campos que serão possíveis realizar somente leitura sem alteração de valores, dessa forma, sendo imutáveis e úteis para utilização em estruturas de código thread safe.

A **Listagem 16** demonstra o uso de readonly em um trecho de código de uma classe, onde será utilizada por uma variável.

O código da **Listagem 16** define uma classe chamada de Cliente que irá possuir um nome e um RG que não poderá ser modificado, representado desta forma pelo uso da palavra readonly. Outras utilizações podem ser definidas como a **Listagem 17** informa.

Listagem 13. Realizando uma sobrecarga

```
type
  Produtos = class
  method BuscaItens(i: Integer): String;
  method BuscaItens(s: String): String;
  property Itens[i: Integer]: String read BuscaItens; default;
  property Itens[s: String]: String read BuscaItens; default;
end;
```

Listagem 14. Utilizando a classe Produtos

```
  var teste : Produtos;
  teste.Itens[5];
  teste[5];
  teste["teste de produto"];
  teste.Itens["teste de produto"];
```

Listagem 15. Definições de arrays

```
type
  meuregistro = record
    caracteres: inline array[0..100] of char;
    inteiros: inline array[1..3,1..5] of integer;
  end;
```

Listagem 16. Uso de readonly

```
type
  Cliente = class
  var Nome : String;
  var RG : String; readonly;
end;
```

Listagem 17. Definição de uma classe como sendo readonly

```
type
  Cliente = public readonly class
  public
    property Nome: string;
    property RG: string;
    constructor(xNome, xRG: string);
  end;
implementation
constructor Cliente(xNome, xRG: string);
begin
  Nome := xNome;
  RG := xRG;
end;
```

Listagem 18. Definição de uma classe e utilização de add e remove

```
type
  MinhaClasseTeste = class
  private
    fMeuEvento: EventHandler;
    method AddMeuEvento(aValue: EventHandler);
    method RemoveMeuEvento(aValue: EventHandler);
  public
    event MeuEvento: EventHandler add AddMeuEvento remove
    RemoveMeuEvento;
  end;
```

Observando as definições da **Listagem 17** é possível verificar que toda a classe Cliente definida será readonly e, para isso, um construtor deverá ser indicado, dessa forma será possível definir um Nome e um RG para o cliente que não poderá mais ser modificado através de código. Assim, ao invés de realizar a alteração de valores dos campos em classes imutáveis, novas classes serão criadas com estes valores.

Add e Remove

Os termos add e remove são utilizados em casos onde quem desenvolve deseja criar a própria lógica de adição e remoção de eventos relacionados, principalmente, com delegates. A **Listagem 18** representa a sua utilização.

Pela **Listagem 18** foi criada uma classe chamada de MinhaClasseTeste e adicionados os métodos AddMeuEvento e RemoveMeuEvento que serão utilizados no evento rotulado de MeuEvento através da utilização de add e remove para melhor exemplificar as suas funcionalidades.

Unsafe

A utilização de unsafe se refere ao uso de ponteiros, ignorando assim a verificação de tipos e segurança garantida formalmente por um código gerenciado. Tal situação poderá ser necessária em cenários de desempenho crítico, principalmente se o aplicativo a ser desenvolvido ou adaptado vier de versões win32 do Delphi para o Delphi Prism onde o uso de ponteiros poderia, por exemplo, ser comumente utilizado, neste aspecto, o Prism prevê o uso de ponteiros.

Além disso, o .NET Framework permite a interoperabilidade entre recursos gerenciados e não gerenciados. A classe Marshal controla como informações são passadas em métodos e retornadas entre as chamadas a objetos gerenciados e não gerenciados. A maioria dos tipos de dados possuem representações comuns tanto na parte gerenciada da memória e da não gerenciada pelo .NET Framework. Para lidar com isso existe a classe Marshal. Além dos tipos comuns, também existem tipos que podem ser ambíguos ou tipos que não possuem uma representação no lado gerenciado. Um tipo ambíguo pode ter representações múltiplas não gerenciadas que apontam para um único tipo gerenciado ou não ter o tipo de informação. Sendo assim, a classe oferece uma coleção de métodos para alocação de memória não gerenciada, cópia de blocos de memória não gerenciados, e conversão de tipos gerenciados em tipos não gerenciados, além de vários outros métodos úteis.

Dessa forma, métodos, propriedades, campos e variáveis locais podem ser definidas como unsafe, bastando para isso adicionar a palavra chave ao final da declaração. A **Listagem 19** demonstra o uso de unsafe na declaração de um método.

Pela **Listagem 19** pode-se observar a definição de uma classe e um método com a palavra unsafe, assim, possibilitando o uso de memória não gerenciada e a declaração da variável inteira de ponteiro x indicada pelo operador circunflexo "^", realizando a definição do ponteiro de inteiro.

Pinned

Variáveis locais podem utilizar o recurso pinned. Isto irá indicar que a memória para a qual ela faz referência não poderá ser removida enquanto a variável estiver presente. Dessa forma, possibilitando que a atribuição para a variável seja única. Utilizando em conjunto com um código unsafe, a **Listagem 20** indica o uso de pinned para a atribuição de uma variável.

Na **Listagem 20** a definição do método Test2 segue os mesmos princípios da definição do método na **Listagem 19**, ou seja, o método deverá ser unsafe para que a variável x possa receber a palavra chave pinned. Além disso, o uso de um array de inteiros também é observado para dar continuidade ao exemplo e indicada a atribuição do endereço do operador através do caractere "@".

Listagem 19. Uso de unsafe na declaração de um método

```
type
  MainForm = class(System.Windows.Forms.Form)
  public
    constructor Create;
    class method Main;
    method Test2; unsafe;
  end;

implementation

method MainForm.Test2;
var
  x: ^Integer;
begin
end;
```

Listagem 20. Indicando um código com pinned

```
Method x.Test2;
begin
  var x: ^Integer; pinned;
  var y: Array of Integer;
  y[0] := 10;
  y[1] := 15;
  x := @y[0];
end;
```

Nota

Para compilar um aplicativo utilizando unsafe, será necessário habilitar uma diretiva de compilação dentro das propriedades do projeto. Na aba Compatibility a diretiva Allow unsafe code deverá ser habilitada.

Conclusão

O Delphi Prism trouxe muitas melhorias e novas funcionalidades para a linguagem, facilitando e acelerando ainda mais a produção de sistemas. Elas ampliam o conhecimento de quem as utilizam e permitem realizar um tratamento mais adequado para a confecção de códigos.

No decorrer deste artigo, foram mencionados alguns dos principais recursos presentes na linguagem, o uso de Delegate, Type, Abstract, Final, Has, Is, Iterator, Of, Default, Inline, Readonly, Add, Remove, Unsafe e Pinned como destaques deste artigo.

Esta sequência de artigos relacionadas às melhorias de linguagem do Delphi Prism tentou destacar o que há de novo e de melhor para o desenvolvimento de sistemas, os novos recursos de linguagem que ampliam o conhecimento e permitem um tratamento mais adequado no fluxo de

código a ser criado. Ajudam a criar códigos mais enxutos e eficazes utilizando recursos até o momento inexistentes em outras versões anteriores do Delphi.

Várias das ideias transmitidas no artigo poderão ser aplicadas no desenvolvimento, principalmente voltadas para questões de desempenho e produtividade.

Rodrigo Araujo
rodrigo.arsi@gmail.com
Bacharel em Ciência da Computação pela UFSM – Santa Maria/RS trabalha com Análise e Desenvolvimento de Software há mais de 10 anos. Desde 2003, utiliza tecnologias .NET.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Site oficial do Delphi Prism
http://www.embarcadero.com/products/delphi-prism

Wiki do Delphi Prism
http://prismwiki.codegear.com/en/Main_Page

Download do Delphi Prism
https://downloads.embarcadero.com/free/delphi_prism

Site oficial do ASP.NET
http://www.asp.net

Visual Studio
www.microsoft.com/visualstudio

Delegate
http://prismwiki.codegear.com/en/Delegate_(keyword)

Type
http://prismwiki.codegear.com/en/Type_(keyword)

Abstract
http://prismwiki.codegear.com/en/Abstract_(keyword)

Final
http://prismwiki.codegear.com/en/Final_(keyword)

Has
http://prismwiki.codegear.com/en/Has_(keyword)

Is
http://prismwiki.codegear.com/en/Is_(keyword)

Of
http://prismwiki.codegear.com/en/Of_(keyword)

Default
http://prismwiki.codegear.com/en/Default_(keyword)

Inline
http://prismwiki.codegear.com/en/Inline_(keyword)

Readonly
http://prismwiki.codegear.com/en/Readonly_(keyword)

Add e Remove
http://prismwiki.codegear.com/en/Remove_(keyword)

Pinned
http://prismwiki.codegear.com/en/Pinned_(keyword)

Unsafe
http://prismwiki.codegear.com/en/Unsafe_(keyword)

Iterator
http://prismwiki.codegear.com/en/Iterator_(keyword)

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre ouutros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Delphi Prism

Entendendo o que são Assemblies, Class Libraries e GAC

Guinther Pauli

Resumo DevMan

De que trata o artigo:

Este artigo apresenta uma visão geral sobre Assemblies, versionamento de DLLs e GAC (Global Assembly Cache) na plataforma .NET, usando o Delphi Prism.

Para que serve:

Um assembly é uma unidade básica gerada pela compilação de aplicações no .NET. Servem para facilmente distribuir módulos e executáveis que podem ser versionados no GAC. O GAC por sua vez serve para terminar com o problema existente de múltiplas versões da mesma DLL rodando no S.O.

Em que situação o tema é útil:

É útil na distribuição de aplicações que fazem uso de assemblies compartilhados, de forma que o CLR (Common Language Runtime) através do GAC gerencia quem deve acessar o que no S.O., independente da localização física dos arquivos no disco.

GAC, Assemblies e Class Libraries:

O objetivo deste artigo é apresentar o GAC (Global Assembly Cache) e mostrar como você desenvolvedor pode tirar proveito dele para instalar e versionar assemblies desenvolvidos para o .NET Framework com Delphi Prism. Através de um exemplo prático desenvolvido no Visual Studio 2010, vamos construir uma Class Library em Delphi Prism e aproveitar para conhecer todos os detalhes envolvidos no processo de distribuição de DLLs no GAC. Antes de começar nosso exemplo, vamos relembrar um pouco sobre DLLs para que você possa entender o porquê do surgimento do GAC.

Acredito que todo desenvolvedor Delphi já precisou, mais cedo ou mais tarde, construir uma DLL. Bibliotecas de vínculo dinâmico (Dynamic Link Libraries) são utilizadas na plataforma Windows há anos e servem para os mais diferentes propósitos: compartilhamento de código binário, criação de extensão de aplicativos e plugins, modularização de aplicações, armazenamento de recursos (como ícones e imagens), biblioteca para uso geral (com funções acessadas por vários aplicativos), desenvolvimento de componentes COM e ActiveX Libraries, entre outros.

No .NET Framework você pode criar uma DLL (chamada de Class Library) para os mais diferentes propósitos: hospedar classes de negócio, criar uma camada independente de acesso a dados, criar bibliotecas de funções e rotinas comuns e reutilizáveis, facilitar a distribuição e instalação de custom controls e componentes etc. Dessa forma, é fundamental que você saiba trabalhar com Assemblies e decidir qual a melhor forma de distribuição. O próprio .NET Framework é distribuído na forma de DLLs compartilhadas. Não fosse isso, cada aplicação precisaria compilar o código do framework, o que aumentaria consideravelmente o tamanho dos arquivos a serem distribuídos. É por isso que aplicações .NET têm alguns poucos Kb. Algo semelhante a compilar um projeto Win32 com a opção "Build with runtime packages".

O Windows internamente faz uso de DLLs, sendo esse o formato para armazenar importantes partes do sistema operacional: kernel32.dll, user32.dll, gdi32.dll são alguns exemplos. Desenvolvedores de aplicativos podem utilizar a API do Windows para se comunicar diretamente com DLLs nativas do Sistema Operacional ou mesmo DLLs específicas, usando DLLImport no caso do .NET. A tecnologia Windows Forms, por exemplo, é uma camada sobre as DLLs Win32 do S.O., nos abstraindo de detalhes específicos e do uso de tipos esotéricos da API, como PChar, Cardinal etc. Exatamente como faz a VCL no Win32.

Como todos sabem, DLLs que precisam ser compartilhadas por vários aplicativos

são colocadas no diretório de sistema do Windows. Aplicações então podem fazer a carga dinâmica ou estática da DLL, localizar um procedimento de entrada exportado, obter um ponteiro para o endereço da rotina e fazer a chamada (isso para DLLs Win32).

As coisas pareciam estar indo bem até que um aplicativo precisasse instalar uma nova versão de uma DLL. Por exemplo, instalamos um software chamado Foo1.exe que faz uso de rotinas armazenadas na biblioteca HellLibrary.DLL, que reside no System32. Fazemos o download de um segundo aplicativo, Foo2.exe, que possui uma versão mais atualizada da DLL e a sobrescreve durante a instalação. Ao executarmos novamente o aplicativo Foo1.exe, podemos ter algumas surpresas desagradáveis, como por exemplo, a mostrada na **Figura 1**.

Listagem 1. Código da classe Calculadora

```
namespace Calc;

interface

uses
  System;

type
  Calculadora = public class
  public
    method Somar(x: System.Int32; y: System.Int32): System.Int32;
  end;

implementation

method Calculadora.Somar(x: System.Int32; y: System.Int32): System.
Int32;
begin
  exit x + y;
end;

end.
```

Listagem 2. Acessando a Class Library (DLL)

```
method MainForm.button1_Click(sender:
System.Object; e: System.EventArgs);
begin
  var x: System.Int32 := Convert.ToInt32
  (textBox1.Text);
  var y: System.Int32 := Convert.ToInt32
  (textBox2.Text);
  var c: Calc.Calculadora := new Calc.Calculadora();
  var r: System.Int32 := c.Somar(x, y);
  label1.Text := r.ToString()
end;
```

Isso pode acontecer porque a nova versão da DLL teve um procedimento alterado, ou renomeado. A menos que o vínculo seja feito pelo índice, a simples alteração do nome de um procedimento fará a aplicação falhar. Em outras situações, a assinatura de um procedimento muda, por exemplo, podendo receber um novo parâmetro. Para que Foo1.exe possa funcionar novamente precisamos recompilar toda a aplicação, o que muitas vezes é inviável. Por esse motivo as interfaces da OO são utilizadas amplamente quando se fala em integrar aplicativos, garantindo contratos imutáveis.

Outro erro comum é quando a própria biblioteca já não existe mais (quem não lembra do erro "Não foi possível encontrar a biblioteca de vínculo dinâmico ..."). Existia uma solução de contorno, muitas vezes era possível "enganar" o Windows simplesmente criando uma DLL vazia (a partir de um arquivo texto mesmo) com o nome da biblioteca que está faltando no disco, e a aplicação poderia então ser executada. Isso porque a carga, nesse caso, era feita estaticamente. A verificação da existência da DLL era feita no momento da inicialização, mas a aplicação poderia nem mesmo fazer uso das rotinas durante a execução.

Esse problema se tornou tão comum no Windows que costumamos chamá-lo de "DLL Hell", ou "Inferno das DLLs". Uma deficiência do Sistema Operacional que levou cerca de 10 anos para ser suprida. Algumas versões mais recentes do Windows resolvem isso mantendo arquivos de instalação que monitoram mudanças nessas DLLs.

Como o .NET acabou com o DLL Hell

Os bons tempos voltaram com o .NET, desde a primeira versão. Um dos princípios básicos do .NET Framework é permitir o deploy por "XCOPY" (é isso mesmo, aquele comando do DOS). Ou seja, para instalar uma aplicação, copie-a para uma pasta. Para desinstalar, apague-a. Nada de chaves de registro, comuns por exemplo no deploy de componentes MTS/COM+. Se uma configuração for necessária, aplicações normalmente vão usar arquivos XML.

O .NET Framework termina com o problema de DLL Hell introduzindo um elegante, robusto e unificado sistema de gerenciamento de DLLs compartilhadas (Assemblies, nesse caso), conhecido como Global Assembly Cache (ou simplesmente GAC). A melhor expressão que encontrei para resumir a função desse importante recurso do .NET é "um System32 com controle de versão". Com ele, aplicações podem usar diferentes versões da mesma DLL instalada no disco (para ser mais exato, no GAC) e não deixarão mais de funcionar caso uma nova versão seja disponibilizada. Se o aplicativo Foo1.exe foi compilado com uma referência à versão 1.0 de HellLibrary.DLL, não falhará caso instalemos Foo2.exe e HellLibrary.DLL 2.0. Ambos podem conviver lado a lado. De fato, no .NET esse comportamento é conhecido como Side-By-Side Execution (veremos como fazer uso dele na prática neste artigo).

Class Library (DLL)

No Delphi Prism XE / Visual Studio 2010, para criar uma Class Library, utiliza-se a opção File>New Project (**Figura 2**). Neste exemplo, a classe chamada "Calculadora" está hospedada no Assembly, que será acessada por uma aplicação cliente Windows Forms.

A implementação da classe criada é mostrada na **Listagem 1**. Como você pode ver, para facilitar, a classe tem uma estrutura bem simples para que possamos nos concentrar nos detalhes específicos ao deploy de assemblies e não na programação da classe em si. Neste ponto é possível dar um Build para gerar a DLL da Class Library.

Uma aplicação Windows Forms usará a classe da calculadora presente no Assembly criado. A referência será feita de forma direta, por enquanto, sem o uso

Nota do DevMan

Interoperabilidade entre linguagens .NET

A integração entre linguagens, componentes ou aplicações nunca foi algo simples. Várias tecnologias surgiram com esse objetivo: o COM (Component Object Model), DDE (Dynamic Data Exchange), OLE (Object Linking and Embedding), ActiveX etc. No Win32, quem nunca precisou acessar uma DLL escrita em outra linguagem? A integração é praticamente rudimentar. Você não podia estender um objeto COM escrito em VB declarando uma classe descendente em Delphi, por exemplo.

Com o .NET isso muda drasticamente. Todas as linguagens feitas para o .NET utilizam os mesmos tipos básicos, através de uma especificação chamada CTS (Common Type System). Por exemplo, os tipos System.String, System.Object, System.IO.File, System.Data.DataSet podem ser utilizados em qualquer linguagem que possua um compilador para .NET. Além disso, a CLS (Common Language Specification) define convenções e regras destinadas a garantir a interoperabilidade entre diferentes linguagens (C#, Delphi Prism, VB.NET etc.). Dessa forma, você pode tirar proveito de todos os recursos oferecidos pela POO entre diferentes linguagens, como herança, tratamento de exceção e polimorfismo, usando somente o código binário (em formato MSIL – Microsoft Intermediate Language) gerado pelo compilador da outra linguagem.

de controle de versão ou de instalação no GAC. Com isso dizemos que nossa Class Library é privada, pois será acessada localmente por uma única aplicação. Mais adiante vamos torná-la pública.

Aplicação para acesso à Class Library

A aplicação Windows Forms pode ser criada usando o menu File>New Project. Neste exemplo seu Name foi definido como "App1". Para adicionar uma referência à Class Library criada anteriormente, no Solution Explorer, utiliza-se a opção Add Reference.

A **Figura 3** mostra o design do formulário principal, com os controles colocados (Button, TextBoxes e Label). O código do evento Click do botão é mostrado na **Listagem 2**. Ele faz acesso à classe Calculadora que está na DLL, criando uma nova instância do objeto e invocando seu método Somar, passando como parâmetro os valores dos TextBoxes. O resultado da soma é mostrado na Label.

Observe como é simples acessar classes de uma DLL e invocar seus métodos. Fazemos a referência a ela como se o código fonte da classe estivesse disponível para o projeto Windows Forms, mas observe que o vínculo é binário. Não precisamos do código PAS da classe, apenas da DLL. Isso pode parecer lógico, mas para aqueles que já desenvolveram DLLs no Win32, devem lembrar como era trabalhoso invocar funções em bibliotecas. Era muito comum exportar rotinas, usar tipos primitivos de dados (inteiros, pchars), além de que precisávamos saber a assinatura da rotina ao importá-la, sendo muito difícil definir tipos (como classes) em DLLs e usá-los em um EXE. No .NET a interoperabilidade entre linguagens (veja quadro) permite o uso de código binário (nesse caso código intermediário - IL) como se fosse fonte. Execute a aplicação e faça o teste.

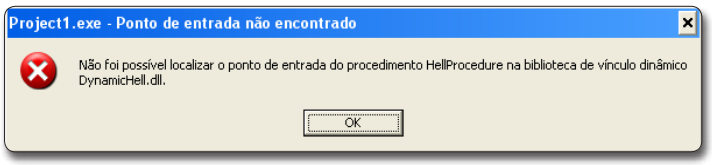


Figura 1. DLL Hell

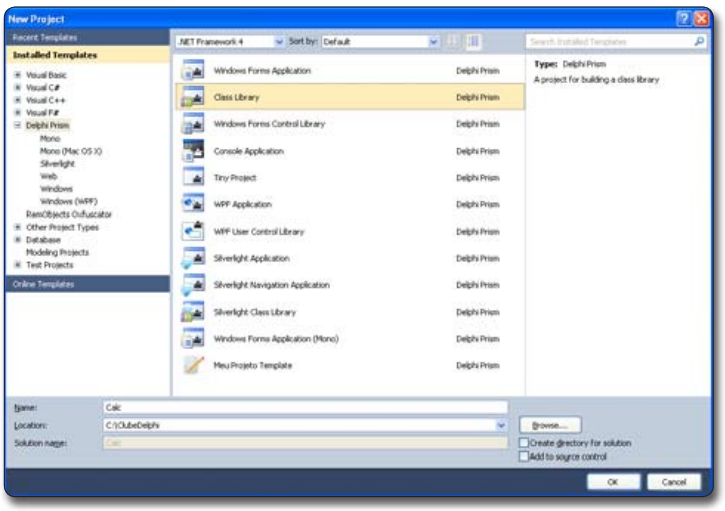
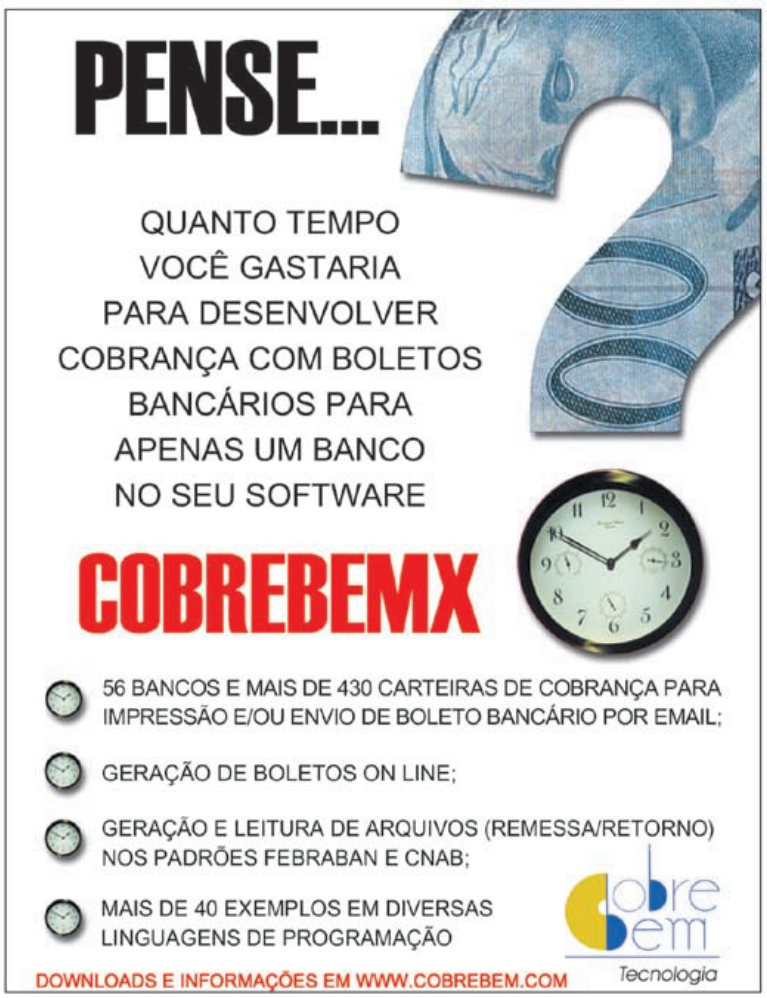


Figura 2. Class Library



Modificando a Class Library

Agora um teste. Na Class Library o código da classe é alterado, por exemplo, modificando-se o nome do método de "Somar" para "Soma" (sem o "r"), seguido de um Build (alteração deve ser desfeita após o teste). Se a aplicação App1 for executada novamente, sem fazer a recompilação da aplicação, um erro semelhante ao da **Figura 4** é mostrado. Esse erro aconteceu porque a aplicação, obviamente, não encontrou o método na Class Library, pois trocamos seu nome. Ou seja, se uma nova versão da DLL for adicionada, aplicações podem deixar de funcionar (acabamos de simular o DLL Hell no .NET).

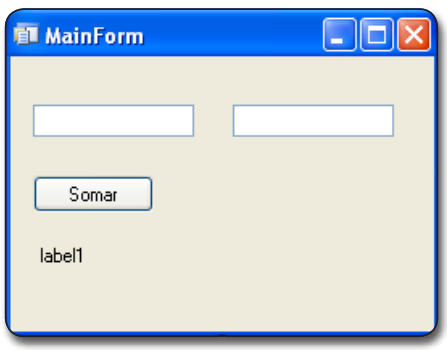


Figura 3. Formulário principal da aplicação

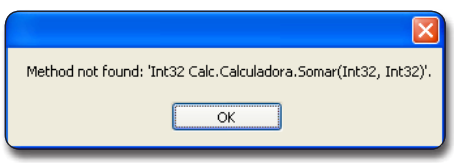


Figura 4. DLL Hell no .NET

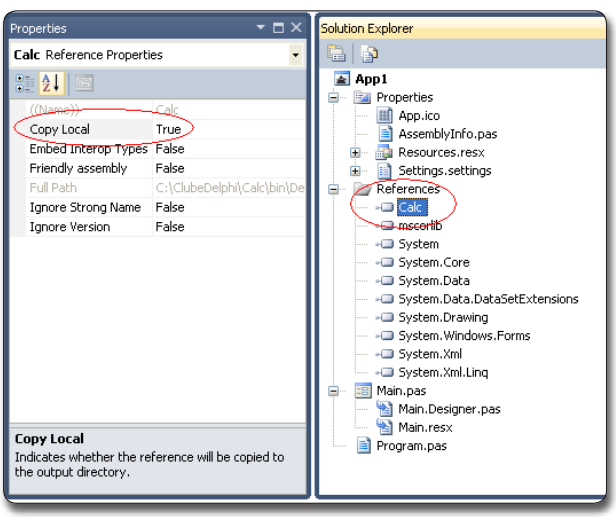


Figura 5. Copy Local

Se o erro não aconteceu no seu teste, provavelmente o Copy Local está habilitado no projeto App1 para o Assembly Calc (**Figura 5**). Com isso, quando a referência à Calc.DLL foi feita, uma cópia da mesma foi colocada no diretório do executável. Dessa forma pode-se copiar manualmente a nova DLL no diretório da Class Library e substituir a que está no diretório da aplicação. É claro, se a aplicação for recompilada com a modificação necessária, ela funcionará normalmente, mas é justamente isso que não queremos fazer neste exemplo para demonstrar os problemas do DLL Hell.

Gerando uma chave pública e assinando o Assembly

Para evitar problemas ocasionados por diferentes versões de Assemblies, ela dever ser colocada no GAC. Ou seja, a DLL poderá agora ser acessada e compartilhada por diferentes aplicações, como se estivesse no System32 do mundo Win32. Além disso, uma vez que esteja no GAC (e não em diretórios privados), podemos fazer uso do controle de versão, sendo que diferentes versões da mesma DLL podem ser executadas ao mesmo tempo.

O primeiro passo é obter uma chave pública (public key) através do utilitário sn.exe (Strong Name) e fazer com que o Assembly seja "assinado" com essa chave. No Visual Studio 2010 isso pode ser feito diretamente de dentro do IDE, para aplicações C#. No Prism, pode-se usar o Visual Studio Command Prompt (**Figura 6**) acessado no menu de programas, com o comando:

```
sn -k CalcKey.snk
```

No projeto da Class Library basta então acessar a opção Project>Project Properties>Signing e informar a referência ao arquivo gerado (**Figura 7**).

Na guia Application é interessante fornecer as informações básicas para o Assembly, como nome. O arquivo AssemblyInfo.pas, que armazena essas configurações, deve estar semelhante ao da **Listagem 3**.

Se o arquivo snk for clicado no Solution Explorer, pode ser visualizado o conteúdo da chave gerada, em formato binário. Ao dar um Build na Class Library, o Assembly será assinado com a respectiva chave gerada e está pronto para ser instalado no GAC.

Instalando o Assembly no GAC

Existem várias formas de instalar um Assembly no GAC. A mais fácil com certeza é usando a extensão do próprio shell do Windows. Para visualizar o GAC basta abrir o diretório c:\Windows\Assembly. Agora simplesmente se arrasta a Calc.dll (a nova, recém compilada) para dentro da janela. Isso seria semelhante a copiar a DLL para o System32 se estivéssemos no mundo DLL Hell Win32.

Uma outra forma mais tradicional de trabalhar com o GAC é através do utilitário gacutil.exe do .NET Framework SDK. A linha de comando a seguir faz o mesmo processo feito anteriormente no shell:

```
gacutil.exe /i Calc.dll
```

Usando Assemblies do GAC

O projeto App1.exe pode ser agora adaptado para que faça uso da Calc.dll que está no GAC, que agora é um Assembly público, ao invés do Assembly privado. No projeto Windows Forms é necessário remover a referência à Calc no Solution Explorer e adicionar uma nova referência ao novo assembly que está no GAC (para o .NET Framework 4.0, o assembly deve estar sob o diretório C:\WINDOWS\Microsoft.NET\assembly\GAC_MSIL\Calc[versão]).

Mais um teste, agora usando o utilitário ildasm.exe do .NET Framework SDK. Ao abrir o arquivo App1.exe (File>Open), em MANIFEST há uma referência para o Assembly externo, Calc, conforme comprova a **Figura 8**. Note que a aplicação contém uma referência explícita para versão da DLL que ela está usando, obtida durante o processo de compilação. Também são listados os demais

Assemblies referenciados pela aplicação, nesse caso os Assemblies do .NET Framework, incluindo o "kernel" do .NET, o mscorlib. Cada assembly possui um manifesto, que é basicamente um conjunto de informações que descrevem os arquivos que o assembly contém e de que outros arquivos ele depende.

Criando uma nova versão da Class Library

Mais um teste. No projeto da calculadora o método Somar agora faz uma operação com 3 números (apenas para efeitos didáticos), como indicado na **Listagem 4**. A **Listagem 5** apresenta a nova versão do AssemblyInfo.pas, note que agora informamos uma nova versão para o Assembly: 2.0.0.1.

Após um Build na Class Library podemos instalar a nova versão no GAC, da mesma forma feita anteriormente. Agora temos uma surpresa: o GAC **não** substituiu a DLL antiga, mantendo em cache as duas versões da mesma, conforme comprova a **Figura 9.** Note que são listadas as versões "1.0.0.1" e "2.0.0.1" da Calc. Isso resolve o problema de substituição de DLLs que existia no mundo DLL Hell e System32: agora múltiplas versões da mesma DLL podem ser compartilhadas e acessadas, convivendo lado a lado em harmonia.

Execução lado a lado (Side-by-Side Execution)

Podemos copiar o diretório completo da aplicação App1 para um diretório chamado App2, sem modificar nem recompilar App1. Ao recompilar o projeto copiado o compilador vai "chiar", pois Calculadora agora espera três argumentos no método Somar.A **Listagem 6** mostra o código adaptado, com mais um TextBox. Observe que tudo funcionária perfeitamente, os três números serão somados e o resultado retornado. Ou seja, a aplicação está fazendo uso da nova versão da Calc.dll.

No idlasm.exe, ao verificar o manifesto de App2, pode ser comprovado que o mesmo faz referência à nova versão da Class Library, a "2.0.0.1.", conforme mostra a **Figura 10**.

Mas e quanto a App1.exe? O comportamento normal, e segundo o DLL Hell, seria o não funcionamento, visto que ela não foi recompilada e o método Somar espera agora não dois parâmetros, mas três. No entanto, o seu manifesto continua apontando para a versão "1.0.0.1" da Calc.dll, de forma que o CLR consegue vincular a versão correta no momento da execução. É só comprovar ao executar App1.exe (sem recompilar). Ela continuará funcionando! É o fim do DLL Hell.

Resumindo:

- App1.exe faz uso da versão "1.0.0.1." da Calc.dll que está no GAC;
- App2.exe faz uso da versão "2.0.0.1." da Calc.dll que está no GAC;

Como existem duas versões da mesma DLL sendo executadas simultaneamente, podemos chamar o recurso de execução

Listagem 3. AssembyInfo.pas

```
namespace Calc;

interface

uses
  System.Reflection,
  System.Resources,
  System.Runtime.InteropServices;

[assembly: AssemblyTitle('Calc ClubeDelphi')]
[assembly: AssemblyDescription('Calc ClubeDelphi')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]
[assembly: AssemblyVersion('1.0.0.1')]
[assembly: NeutralResourcesLanguage('')]
[assembly: ComVisible(false)]
...
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('CalcKey.snk')]
[assembly: AssemblyKeyName('')]

implementation

end.
```

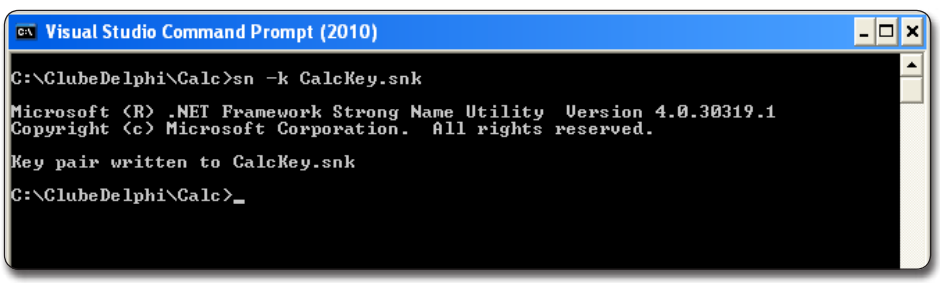


Figura 6. Gerando uma chave para o Assembly

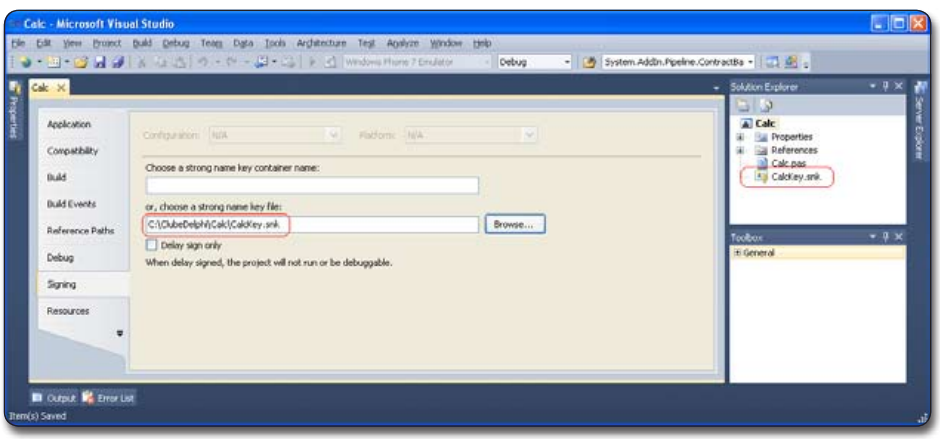


Figura 7. Assinando o Assembly com a chave

Nota do DevMan

CLR é o ambiente de execução - Runtime - no qual toda aplicação feita para plataforma .NET é executada. Fornece uma série de serviços para que sua aplicação consiga ser executada adequadamente nos sistemas operacionais / plataformas de hardware para o qual a plataforma .NET tenha uma versão disponível, como Windows, Mac e Linux (através do Mono). Um termo que nós sempre vamos ler ou ouvir quando estamos tratando do CLR é o termo Managed. Uma aplicação para a plataforma .NET é chamada de Managed Application. Em uma tradução livre, podemos dizer que é uma Aplicação Gerenciada. Mas o que isto quer dizer?

Vamos lembrar o que não é uma Aplicação Gerenciada. Quando você tem um compilador que gera código nativo, este código gerado é específico para um processador ou família de processadores, como Intel x86, por exemplo. Se você está utilizando um compilador, como C/C++, o resultado da compilação do código-fonte, no caso da plataforma Windows de sistemas operacionais, é um arquivo binário/executável, que é reconhecido pelas extensões EXE e DLL. As instruções que são geradas pelo compilador e são gravadas neste arquivo binário/executável não são controladas pelo sistema operacional durante a execução. Isto significa que o sistema operacional não gerencia a execução deste arquivo. O fato de não gerenciar a execução quer dizer que seu binário/executável pode executar praticamente qualquer operação. É claro que recentemente temos tido avanços nos sistemas operacionais mais populares para reduzir certos problemas. Mas hoje, esta responsabilidade em gerar código com mais qualidade e com menos possibilidades de erros está, além é claro do profissional que escreve o código, com os compiladores/linkers. Em muitos casos, os compiladores reconhecem padrões de código e/ou estruturas e inserem códigos para reduzir ou até mesmo evitar determinadas situações. Um erro muito comum em código é tentar acessar uma posição de memória que não está disponível ou não foi inicializada adequadamente para o propósito esperado pelo aplicativo. Neste caso, muitos compiladores inserem códigos, que realizam testes para reduzir estas situações ou até mesmo evitar que o aplicativo continue sua execução, caso esta situação seja encontrada. Como podemos visualizar, é um grande problema para aplicativos/sistemas avançados garantirem a qualidade de código e uma execução mais estável e previsível.

A ideia de um ambiente gerenciado não é nova. De maneira objetiva, um ambiente gerenciado quer dizer que existe um modelo que define uma série de aspectos técnicos e de execução (leia-se comportamento) e que as aplicações devem seguir. Não é caso de um aplicativo com código nativo. Se o código nativo tentar executar algo ilegal, você só vai saber quando visualizar os resultados daquela operação inapropriada. Quem já se deparou com uma aplicação abortando sua execução, aparentemente sem motivo, tem um exemplo prático desta situação.

O CLR fornece um ambiente de execução gerenciado, independente de linguagem de programação. Vamos conversar sobre ser independente de linguagem de programação. Quando compilamos o código-fonte para a plataforma .NET, por exemplo em Delphi Prism, o resultado desta compilação não é um código em Assembly nativo. O compilador gera um arquivo binário/executável no qual o seu código-fonte é compilado em uma linguagem intermediária independente de plataforma. Esta linguagem é a CIL - Common Intermediate Language, já comentada neste artigo. Ela também é chamada de MSIL - Microsoft Intermediate Language em muitas literaturas. A CIL é o que podemos chamar de "linguagem de baixo nível" para a plataforma .NET, mas na realidade ela é muito poderosa e independente de linguagem de programação/plataforma de processador.

O arquivo binário resultante de uma compilação em .NET é chamado de Managed Module, que garante independência de linguagem de programação. Apesar de existirem códigos escritos em linguagens de programação completamente diferentes, todas elas geram instruções em CIL, conforme especificado e suportado pelo CLR. Como parte do CLR, existem componentes que lêem as instruções CIL do arquivo binário/executável e realizam todo um processo de validação, tanto da estrutura física do arquivo que contém o código, quanto das operações que as instruções CIL pretendem executar. Por exemplo, se uma determinada instrução CIL foi gerada pelo compilador da linguagem Delphi Prism, e esta instrução recebe parâmetros incorretos, o CLR, através de seus componentes, impede que este código seja carregado, a aplicação nem chega a ser executada. Existe um processo muito rigoroso para validação do CIL gerado pelos compiladores para a plataforma .NET.

A partir do código-fonte do seu aplicativo, as ferramentas de compilação e de geração de arquivos, criam arquivos binários como resultado destes processos. No Microsoft Windows, o formato de arquivo PE/COFF (Portable Executable and Common Object File Format) é o padrão utilizado para os arquivos do tipo EXE e DLL. Este formato é utilizado nas versões 32-bit e 64-bit dos sistemas operacionais da Microsoft.

Os códigos compilados para o CLR e os recursos que compõem o software são armazenados em uma estrutura física (arquivo PE/COFF) descrita como Managed Module. Em um Managed Module é possível armazenar vários tipos de informação além do código CIL gerado pelo compilador, como um arquivo no formato PE/COFF, cujo conteúdo pode ser executado pelo CLR, além de Metadados e Recursos (Resources).

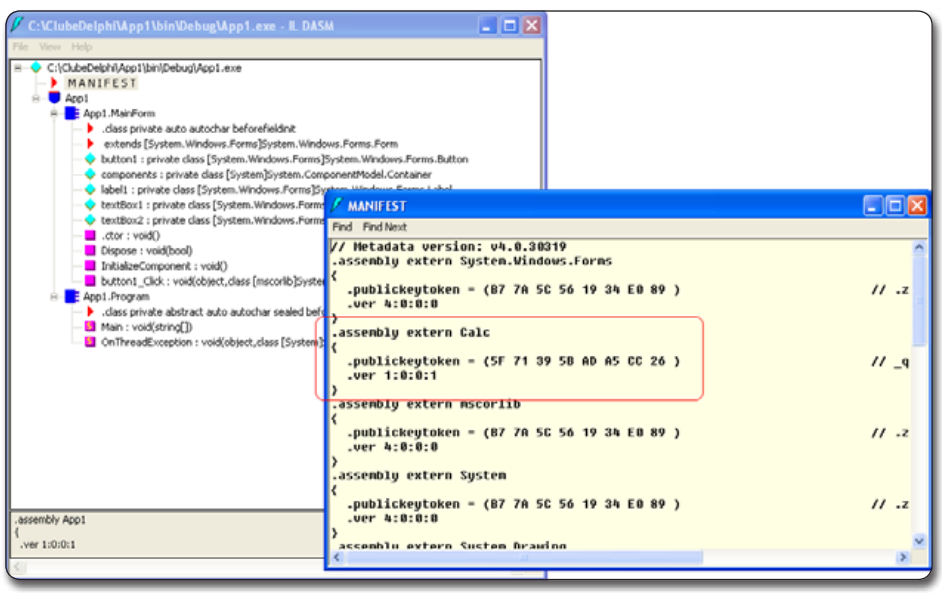


Figura 8. ILDASM indicando o manifesto da aplicação, referenciando a versão 1.0.0.1 da Calc

Listagem 4. Nova versão da Calculadora

```
namespace Calc;

interface

uses
  System;

type
  Calculadora = public class
  public
    method Somar(x: System.Int32; y: System.Int32;
    z: System.Int32): System.Int32;
  end;

implementation

method Calculadora.Somar(x: System.Int32; y:
System.Int32; z: System.Int32): System.Int32;
begin
  exit x + y + z;
end;

end.
```

Listagem 5. Arquivo AssemblyInfo.pas guarda informações do assembly, como versão

```
namespace Calc;
interface
uses
  System.Reflection,
  System.Resources,
  System.Runtime.InteropServices;
[assembly: AssemblyTitle('Calc ClubeDelphi')]
[assembly: AssemblyDescription
('Calc ClubeDelphi')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]
[assembly: AssemblyVersion('2.0.0.1')]
[assembly: NeutralResourcesLanguage('')]
[assembly: ComVisible(false)]
...
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile('CalcKey.snk')]
[assembly: AssemblyKeyName('')]
implementation
end.
```

Listagem 6. Código para acesso à nova implementação do método Somar

```
method MainForm.button1_Click(sender: System.Object; e: System.EventArgs);
begin
  var x: System.Int32 := Convert.ToInt32(textBox1.Text);
  var y: System.Int32 := Convert.ToInt32(textBox2.Text);
  var z: System.Int32 := Convert.ToInt32(textBox3.Text);
  var c: Calc.Calculadora := new Calc.Calculadora();
  var r: System.Int32 := c.Somar(x, y,z);
  label1.Text := r.ToString()
end;
```

lado a lado, ou Side-by-Side Execution, como é conhecido no .NET Framework. A CLR (veja Nota do DevMan) faz todo o controle sobre os Assemblies, compilação em runtime e gerenciamento de memória.

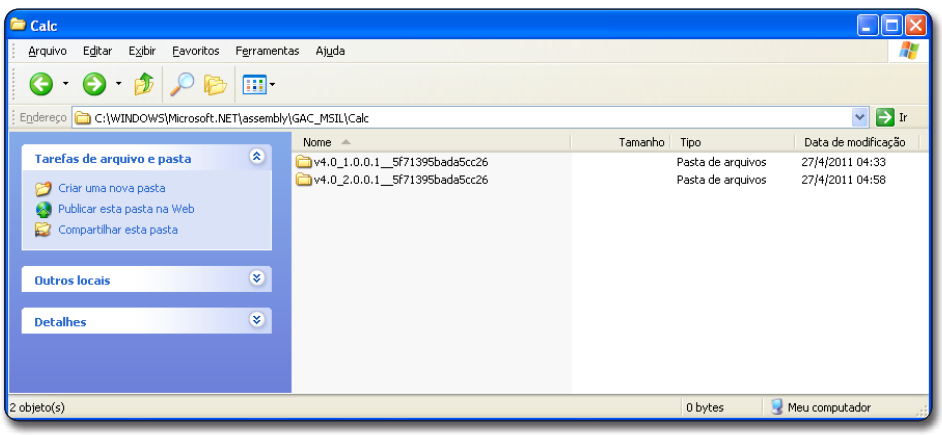


Figura 9. GAC armazena duas versões da mesma DLL

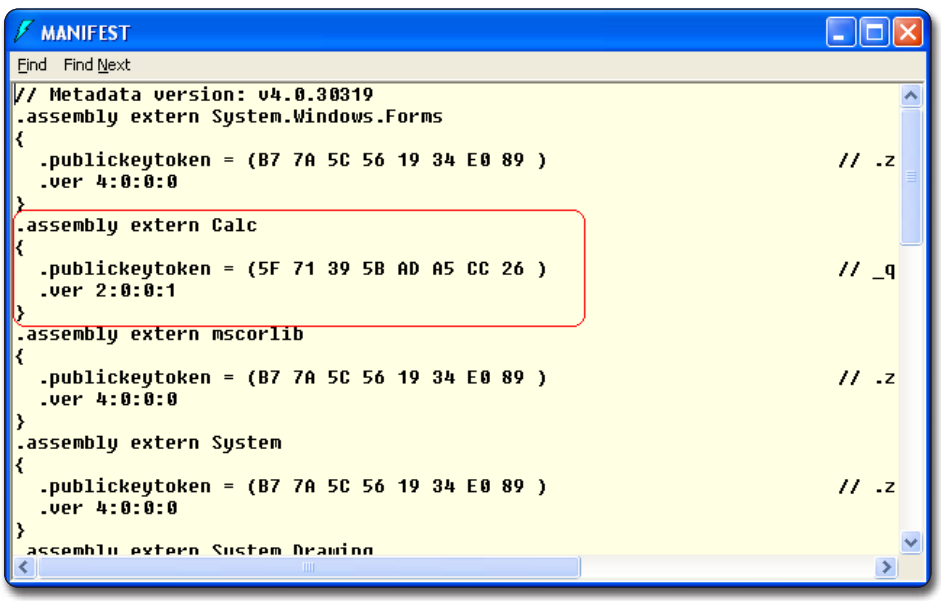


Figura 10. Aplicação faz referência à versão 2.0.0.1 da Calc.dll

Conclusão

Neste artigo conhecemos o GAC e vimos como tirar proveito dele para versionar Assemblies construídos para o .NET Framework. O GAC termina com um antigo problema do Sistema Operacional Windows, o "DLL Hell". E o melhor de tudo: usando o Delphi Prism e Visual Studio, vimos como é extremamente simples fazer uso das técnicas aqui apresentadas.

Guinther Pauli
guinther@devmedia.com.br
guinther.pauli@gmail.com
http://gpauli.com
http://twitter.com/guintherpauli

Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET). Editor da Revista ClubeDelphi e da Revista .NET Magazine Brasil, professor de Pós-Graduação da UNIVEM-SP. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adoção de melhores práticas. Foi arquiteto e desenvolvedor do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

dbExpress
Conceitos relacionados a transações

No mundo corporativo, a utilização de sistemas gerenciadores de banco de dados SGBD é muito comum, praticamente todas as empresas que possuem sistemas fazem uso de banco de dados. Existem diversas opções de sistemas gerenciadores de banco de dados, desde pequenos sistemas que rodam em computadores pessoais até sistemas enormes rodando em mainframes.

É possível classificar os sistemas gerenciadores de banco de dados segundo o seu modelo, dentre eles pode-se destacar o modelo relacional e o orientado a objetos. Ambos possuem pontos positivos e negativos, mas o modelo relacional é sem dúvida o modelo mais utilizado no ambiente corporativo.

O modelo relacional foi desenvolvido por Codd em 1970 no laboratório da IBM. Este modelo também é conhecido como modelo entidade-relacionamento, sendo baseado na percepção do mundo real. Essa ideia consiste em um conjunto de objetos denominados entidades e nos relacionamento entres esses objetos (Korth & Silberschatz, 1995).

Os recursos disponíveis nos gerenciadores de banco de dados se bem utilizados, permitem ao desenvolvedor melhorar o desempenho da aplicação e a confiabilidade das informações fornecidas.

Conceito Transacional

O conceito de transação é fundamental em todos os sistemas gerenciadores de banco de dados. O ponto principal de uma transação é permitir a execução de várias ações em uma única operação que será concretizada se nenhum problema for encontrado ou tudo será desfeito se ocorrer algum problema. Pode-se dizer que uma transação é um processo de tudo ou nada.

Resumo DevMan

De que trata o artigo:

O artigo trata dos conceitos relacionados a transações em servidores de banco de dados relacionais, seus níveis de isolamento e de práticas que garantem um bom desempenho durante a execução de transações, o uso da tecnologia dbExpress para acesso a servidores de banco de dados, detalhando os seus componentes.

Para que serve:

O dbExpress pode ser utilizado para acesso a inúmeros bancos de dados como Oracle, PostgreSQL, Firebird, Mysql, SQLServer, dentre outros. Seus componentes permitem ao desenvolvedor atingir um alto grau de desempenho nas conexões com servidores de banco de dados. O entendimento dos conceitos transacionais facilitará a implantação de processos críticos, que devem ser controlados para garantir a integridade e consistência dos dados.

Em que situação o tema é útil:

Em qualquer sistema que necessite de um controle transacional e que esteja usando dbExpress e um banco de dados relacional.

A tecnologia dbExpress - Utilização e Transações:

No decorrer do artigo vamos entender como utilizar a tecnologia dbExpress para acessar dados de modo simples e eficiente, conhecer suas principais características e componentes, conhecer os conceitos transacionais de servidores de banco de dados relacionais, seus níveis de isolamento, como utilizar o dbexpress para trabalhar com os servidores de banco de dados, entender as mudanças implementadas na utilização de transações com o dbExpress e construir um exemplo que fará uso dos conceitos abordados no artigo.

Durante uma transação os estados intermediários das operações não são vistos por outras transações simultâneas, se ocorrer algum problema que impeça a transação de chegar até o final da execução, as modificações efetuadas durante a transação serão desfeitas e não ocorrerá nenhuma mudança no banco de dados.

Outra característica importante nos gerenciadores de banco de dados transacionais é que ele deve garantir que todas as atualizações realizadas por uma transação fiquem registradas de modo permanente após a transação ser considerada completa, mesmo se ocorrer uma pane em seguida.

As transações estão muito ligadas à noção de atualizações atômicas (indivisível), portanto as transações devem ser tudo ou nada não somente no que diz respeito ao efeito permanente no banco de dados, mas também em termos de visibilidade durante o seu processamento.

De maneira geral, determinam-se os conceitos de uma transação através da sigla ACID (Atomicidade, Consistência, Isolamento, Durabilidade).

- Atomicidade: Tudo e feito ou nada é feito;
- Consistência: Garantia de integridade dos dados, a transação tem que respeitar as regras impostas pelo SGBD;
- Isolamento: As transações concorrentes jamais verão os estados intermediários de outras transações. Caso duas transações simultâneas tentem acessar os mesmos

em servidores de BD relacionais

Jéfferson Ricardo Zuchi

dados, a segunda aguardará o término da primeira, dependendo do nível de isolamento definido para a transação;
• Durabilidade: Após a confirmação de uma transação, seus efeitos são permanentes mesmo em caso de pane posterior.

Níveis de Isolamento Transacional

As transações especificam um nível de isolamento, que define o grau em que a transação deve ser isolada contra as modificações feitas por outras transações. Os níveis de isolamento determinam quais efeitos colaterais são permitidos em caso de simultaneidade.

A maioria das aplicações pode ser desenvolvida sem a necessidade de níveis de isolamento elevados, nível SERIALIZABLE, reduzindo assim o nível de locks ou travamento de registros. Cabe ao desenvolvedor analisar cuidadosamente o código e definir qual é o melhor nível de isolamento para cada tipo de transação. Quanto maior o nível de isolamento maior a probabilidade de bloqueio nos registros.

Os níveis de isolamento definidos pelas normas do ANSI/SQL são:
• SERIALIZABLE: O mais elevado nível de isolamento, esse nível, determina ao gerenciador de banco de dados que todas as transações devem ocorrer de forma totalmente isolada, ou seja, todas as operações serão feitas de forma sequencial, uma após a outra. O isolamento neste nível e tamanho que até mesmo a utilização de select com alguma clausula where causa lock de registros;
• REPEATABLE READS: Este nível de isolamento vê apenas os dados efetivados antes do início da transação, ela não tem acesso a dados não confirmados de outras transações ou mudanças efetivadas por outras transações durante o período em que a transação corrente estiver em execução. Desta forma, diferente do nível Read Commited, sucessivos comandos select sempre trarão os mesmos dados durante a transação corrente;
• READ COMMITED: Normalmente a maioria dos sistemas gerenciadores de banco de dados utiliza esse nível de isolamento como padrão. Neste nível, a transação tem acesso apenas a dados efetivados, nunca enxerga dados que ainda não foram confirmados em outras transações simultâneas. É importante destacar que dois comandos select sucessivos podem recuperar dados diferentes, mesmo que eles estejam em uma única transação, isso pode ocorrer, pois existe a possibilidade de outras transações efetivarem seus dados após a execução do primeiro comando select;
• DIRTYREAD/READ UNCOMMITED: Também conhecida como leitura suja é um nível de isolamento inferior. Neste nível é permitido à transação corrente ver dados ainda não confirmados por outras transações simultâneas.

Com base nos níveis de isolamento podemos resumir a ocorrência dos locks conforme mostrado na **Tabela 1**.

A tecnologia dbExpress

O dbExpress foi desenvolvido para substituir o antigo BDE (Borland Database Engine). Lançado pela primeira vez com a versão 6 do Delphi, ele sofreu uma grande evolução, sendo que a versão 4 está disponível no Delphi 2010.

A arquitetura do dbExpress define uma camada independente através de uma interface comum para fornecer acesso rápido aos servidores de banco de dados SQL. Para cada servidor suportado, o dbExpress provê um drive com uma biblioteca independente que implementa suas interfaces comuns para processamento de consultas e procedimentos de armazenamentos. Como a camada de acesso a dados é pequena e simples de implementar, existem drivers para diversos gerenciadores de banco de dados de alto desempenho, tanto proprietários quanto open source. Os drivers são fornecidos pela própria Embarcadero ou por terceiros através de um arquivo DLL que deve ser distribuído juntamente com a aplicação.

Principais características do dbExpress

O dbExpress é multiplataforma, atua tanto nas plataformas Windows, quanto nas inúmeras versões do Linux através do Kylix que foi descontinuado pela Borland. Por ser uma camada muito "fina" sobre a API de banco de dados subjacente adiciona pouco overhead às operações de banco de dados. Consequentemente, com pouco overhead, o dbExpress apresenta um grande desempenho. Foi projetado para trabalhar com o ClientDataset do Delphi.

Na distribuição da aplicação é necessária a inclusão do driver dbExpress correspondente ao SGBD que será conectado, contido em um arquivo DLL e do arquivo midas.dll, assim, oferece um deploy simplificado. Fazer uso das interfaces para implementar a camada de persistência, facilita a mudança de um gerenciador de banco de dados para outro, uma vez que os métodos contidos na interface são idênticos para qualquer sistema gerenciador de banco de dados. É possível desenvolver um sistema inicialmente projetado para Firebird e migrá-lo para Oracle de maneira transparente ou com o mínimo de modificações.

Hoje existem diversos gerenciadores de banco de dados suportados pela tecnologia dbExpress, através do uso de drivers, muitos desenvolvidos pela própria Embarcadero. Também é possível encontrar drivers dbExpress pagos, desenvolvidos por empresas

terceiras e opções open source. Alguns dos principais gerenciadores de banco de dados que possuem drivers disponíveis para dbExpress são: Oracle, Firebird, InterBase, DB2, Informix, SQL Server, MySQL etc.

Componentes do dbExpress

Os componentes da paleta dbExpress permitem a conexão com diversos servidores de banco de dados. Nessa paleta temos:

TSQLConnection: Responsável por estabelecer a conexão diretamente com sistema gerenciador de banco de dados. É neste componente que vamos configurar qual o caminho do banco de dados, usuário, senha, e qual é o servidor que se deseja estabelecer a conexão. A **Figura 1** mostra as configurações necessárias para a conexão com um banco de dados Firebird. Praticamente todas as informações serão preenchidas automaticamente após a seleção da propriedade Driver no Object Inspector, cabendo apenas ao desenvolvedor indicar os valores para a linha Database, User_Name e Password. Nota-se que a propriedade VendorLib é o arquivo DLL distribuído juntamente com o Firebird, e a linha LibraryName contém o nome do driver dbExpress que implementa as rotinas de acesso ao banco de dados.

TSQLTable: Este componente é utilizado para representar uma tabela do banco de dados. Na verdade, este componente gera uma instrução SQL que recupera todos os dados contidos na tabela especificada. Para ter acesso ao banco de dados ele faz uso do componente TSQLConnection. É um componente unidirecional, que não faz buffer dos registros na memória, por esse motivo, não dá suporte completo à navegação de dados, sendo somente possível utilizar os comandos First e Next. Para possuir acesso completo ao recurso de navegação de dados o componente TSQLTable deve ser utilizado em conjunto com o componente TDataSetProvider e o componente TClientDataSet.

TSQLQuery: Este componente é muito similar ao componente TSQLTable, porém ele permite a execução de instruções SQL no servidor de banco de dados, no caso da instrução SELECT o TSQLQuery conterá as linhas e as colunas referentes à resposta do servidor de banco de dados. Além da instrução SELECT, também é possível executar as instruções INSERT, DELETE, UPDATE e outras mais.

TSQLDataSet: Este componente é o mais genérico dentre os componentes dbExpress. O TSQLDataSet é conectado diretamente no componente TSQLConnection e pode atuar como um TSQLQuery ou TSQLTable e como um TSQLStoredProcedure. Isso é possível devido às propriedades CommandType e CommandText do componente. A propriedade CommandType determina como ele deve ser instanciado, se for definido ctQuery, a propriedade CommandText será utilizada para rodar comandos SQL. Se for definido ctTable, a propriedade CommandText especificará o nome de uma tabela do banco de dados. Finalmente se ctStoredProcedure for definido, CommandText especificará o nome da stored procedure que será chamada.

Os componentes TSQLDataSet, TSQLTable e TSQLQuery, são componentes unidirecionais, isso significa que eles não conseguem enviar modificações efetuadas localmente para o servidor. Para que isso seja possível o desenvolvedor deve utilizar os componentes TDataSetProvider e TClientDataSet em conjunto com um dos outros citados acima para utilizá-los de modo bidirecional, isso significa que será possível enviar as modificações para o servidor, bem como possuir suporte completo à navegação de dados. Na maioria dos casos, em ambiente corporativo utiliza-se normalmente esta combinação TSQLDataSet, TDataSetProvider e TClientDataSet.

TSimpleDataSet: Este componente incorpora os componentes TSQLDataSet e o TDataSetProvider em um único componente, assim é possível utilizá-lo diretamente conectado ao TSQLConnection e efetuar operações de leitura e enviar as alterações efetuadas localmente para o servidor através da utilização do método ApplyUpdates, da mesma forma que o componente TClienteDataSet. Trabalhando de modo bidirecional ele proporciona a criação de buffer de dados em memória permitindo ao desenvolvedor utilizar os comandos Post e Delete do próprio componente e também dá suporte total à navegação de dados. Este componente foi desenvolvido para desenvolvimento de aplicação em duas camadas

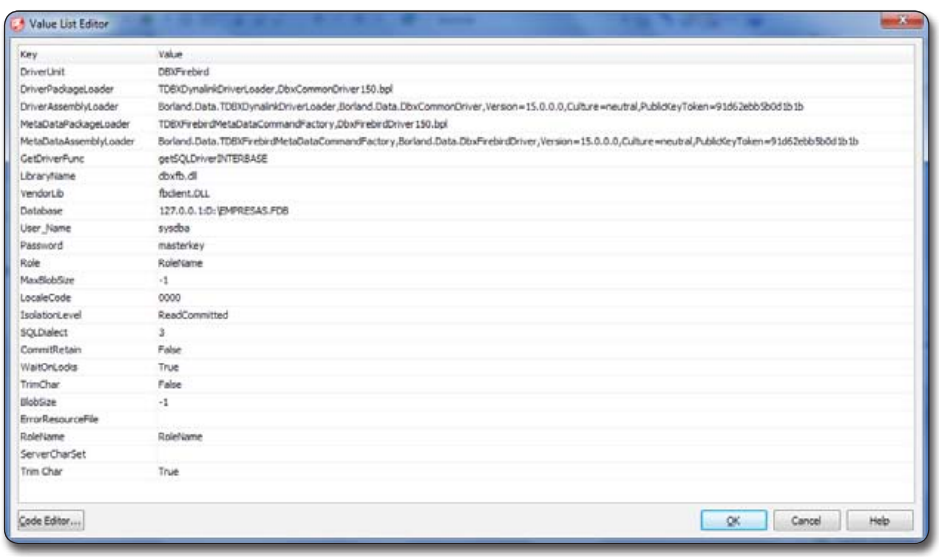


Figura 1. Parâmetros para conexão

Nível de Isolamento	Bloqueio por Faixa	Bloqueio na Leitura	Bloqueio na Gravação
Serializable	S	S	S
Repeatable Reads	N	S	S
Read Commited	N	N	S
DirtyRead/Read Uncommited	N	N	N

Tabela 1. Locks vs. nível de isolamento

não sendo recomendado para utilização em operação unidirecional.

Utilizando os controles dbExpress

O formulário da **Figura 2** permite ao usuário pesquisar informações no banco de dados através da data de vencimento, bem como incluir novos registros e alterar dados utilizando o componente TSimpleDataSet. Este exemplo demonstra como podemos utilizar esse componente de maneira rápida e simples.

A **Listagem 1** possui dois métodos, o mais importante é entender que o método sdsContasReceberBeforePost(DataSet: TDataSet) sempre será executado antes de ocorrer um evento de gravação, e o método newId encapsula a lógica para a geração de um novo id que será utilizado como chave primária da tabela contas receber.

O código a seguir demonstra a utilização do comando ApplyUpdates, responsável pelo envio das atualizações feitas localmente para o servidor de banco de dados. Este método será executado toda vez que ocorrer um Post. Esse mesmo código também deve ser executado toda vez que o usuário excluir um registro:

```
procedure TForm1.sdsContasReceberAfterPost
(DataSet: TDataSet);
begin
  sdsContasReceber.ApplyUpdates(-1) ;
end;
```

A utilização de campos calculados é uma alternativa à utilização de campos de lookup. A vantagem desta abordagem é que não é necessário trazer para a memória do computador cliente a lista de todas as empresas. A desvantagem é que serão feitas diversas chamadas ao servidor de banco de dados para recuperar individualmente o nome de cada uma das empresas. Cabe ao desenvolvedor analisar qual das alternativas melhor se adapta a sua necessidade. O código a seguir mostra a utilização desta abordagem:

```
procedure TForm1.sdsContasReceberCalc
Fields(DataSet: TDataSet);
begin
  if sdsContasReceberEMP_ID.IsNull then
    sdsContasReceberCLIENTE.AsString :=''
  else
    sdsContasReceberCLIENTE.AsString := getNomeEm-
presa( sdsContasReceberEMP_ID.AsInteger );
end;
```

A **Listagem 2** é responsável por efetuar a pesquisa no banco de dados, um detalhe importante é a utilização de parâmetros que são atribuídos ao SQL por meio do uso da propriedade ParamByName. Utilizando essa abordagem não precisamos nos preocupar com o formato da data, pois o mesmo será tratado diretamente pelo driver dbExpress indicado no componente. Se por algum motivo a aplicação precisar ser migrada para outro banco de dados o impacto de migração será muito menor. É óbvio que outros cuidados também devem ser tomados para minimizar possíveis problemas no caso de desenvolvimento para múltiplos SGBDs. É importante salientar que o componente TSimpleDataSet utiliza um InternalDataSet que pode ser usado como um TSQLTable ou TSQLQuery e como um TStoredProcedure. Neste caso estamos utilizando-o como uma query, conforme defino na linha 07. Outro detalhe é que atribuímos o comando SQL na propriedade DataSet, mas deve-se executar o comando open do próprio componente TSimpleDataSet para que ele possa gerenciar e controlar os dados retornados do servidor.

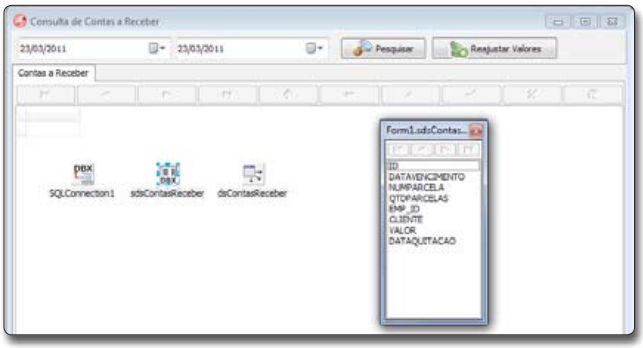


Figura 2. Tela de pesquisa e manutenção utilizando o TSimpleDataSet

Transações com DBExpress

Existem duas formas ainda suportadas pelo dbExpress para utilizar os recursos transacionais dos gerenciadores de banco de dados. A primeira forma é utilizada até a versão 3 do dbExpress, a partir da versão 4 a sintaxe foi modificada ficando mais intuitiva e fácil de utilizar.

Ainda é possível utilizar a versão antiga, mas isso é muito

Listagem 1. Método para gerar um novo id

```
function TForm1.newId: Integer;
var
  query : TSQLQuery ;
begin
 query := TSQLQuery.Create( self );
 query.SQLConnection := SQLConnection1 ;
 try
  query.SQL.Text := ' select max(id)+1 from contasreceber ' ;
  query.Open ;
  // testa se o select retornou null
  if Trim(query.Fields[0].AsString) = EmptyStr then
   result := 1
  else
   result := query.Fields[0].AsInteger ;
 finally
  query.Close ;
  FreeAndNil( query );
 end;
end;

procedure TForm1.sdsContasReceberBeforePost(DataSet: TDataSet);
begin
  // em caso de inserção atribui um id para novos registros
  if dataSet.State = dsInsert then
    sdsContasReceberID.AsInteger := newId() ;
end;
```

Listagem 2. Método responsável pela pesquisa de contas a receber

```
01 procedure TForm1.spbPesquisarClick(Sender: TObject);
02 begin
03   with sdsContasReceber do
04   begin
05     Close ;
06     DataSet.Close ;
07     DataSet.CommandType := ctQuery ;
08     DataSet.Params.Clear ;
09     DataSet.CommandText := ' SELECT * FROM CONTASRECEBER ' +
10           ' WHERE DATAVENCIMENTO BETWEEN :dtInicial
               AND :dtFinal ' +
11           ' ORDER BY DATAVENCIMENTO, ID ';
12     DataSet.ParamByName('dtInicial').AsDate := DateTimePicker1.
       Date ;
13     DataSet.ParamByName('dtFinal').AsDate := DateTimePicker2.
       Date ;
14     Open;
15   end;
16 end;
```

desaconselhável visto que ela só foi mantida para garantir compatibilidade e a tendência é que ela seja removida em versões futuras.

As versões anteriores à versão 4 do dbExpress fazem uso de um record, o TTransactionDesc, que deve conter as características da transação que será inicializada pelo servidor de banco de dados. Deve-se dar uma atenção especial a dois elementos deste record, são eles o **TransactionID**, responsável por conter o id único que é utilizado como identificador da transação e o elemento **IsolationLevel,** responsável por determinar qual será o nível de isolamento da transação. Esses valores estão disponíveis através do tipo TTransIsolationLevel. Os valores válidos são xilREADCOMMITTED, xilREPEATABLEREAD, xilDIRTYREAD, xilCUSTOM. Para utilizá-los basta adicionar na cláusula uses do projeto a unit SqlExpr.

Também são usados no código da **Listagem 3** os comandos Ramdomize e Random. O primeiro é responsável pela criação de uma semente de números aleatórios e o comando Random, seleciona aleatoriamente um número dentro do range especificado, gerando assim um id único para a transação que será inicializada. Vale lembrar que o valor zero não é um valor válido para o elemento TransactionID. Após adquirido o id da transação e determinado o seu grau de isolamento, utiliza-se a procedure StartTransaction implementada na classe SQLConnection para iniciar uma nova transação. O método StartTransaction receberá como parâmetro as definições da transação que foram definidas pelo record TTransactionDesc.

É uma boa prática utilizar o bloco try except para garantir o rollback caso algum problema ocorra durante o processamento da transação. No exemplo a seguir qualquer exception que ocorrer durante o bloco transacional será tratado e o rollback será executado, desfazendo todas as alterações processadas durante a transação. Se todo o processo for efetuado com sucesso, a procedure commit será chamada e efetivará todas as mudanças que ocorreram no banco de dados.

A **Listagem 4** mostra a aplicação do controle transacional utilizando o modelo do dbExpress 4. Não é mais utilizado diretamente um record para determinar as características da transação, mas sim um método que encapsula essa complexidade. O método BeginTransaction passa a ser utilizado como uma fábrica de objetos TDBXTransaction. A grande vantagem dessa mudança é que toda a lógica para a criação de uma transação fica encapsulada e as modificações só afetarão os projetos desenvolvidos caso ocorra mudança na assinatura da função BeginTransaction, o que é muito pouco provável. A classe TDBXTransaction está definida na unit DBXCommon.

O botão reajustar valores, **Listagem 5**, terá a responsabilidade de atualizar o valor da contas a receber com base em uma porcentagem indicada pelo usuário e registrar a atualização na tabela de histórico. Esse procedimento ocorrerá dentro de uma transação, se ocorrer qualquer problema todo o processamento será revertido. Apesar de ser um exemplo simples é possível demonstrar boas práticas para a utilização desse recurso indispensável nos sistemas de gerenciamento de banco de dados.

Foram definidas variáveis locais, inclusive uma componente do tipo TSQLQuery. Elas só serão alocadas durante a execução do procedimento, isso economiza memória e deixa o código mais simples de manter.

Toda a validação de dados e inputs do usuário é feita antes de iniciar a transação, isso porque é muito importante minimizar ao máximo o tempo de execução de uma transação. Uma transação consome muitos recursos do servidor, e quanto mais rápida for sua execução menos recursos o servidor manterá alocado para ela. Após garantir a entrada de dados e a validação, iniciamos a transação e executamos o comando update responsável pela atualização de valores. Outros comandos também podem ser

Listagem 3. Modelo de controle transacional com dbExpress 3 e anteriores

```
procedure TForm1.Rotina1(Sender: TObject);
var
  TD : TTransactionDesc ;
Begin
  Randomize();
  TD.TransactionID := Random(2000) + 1 ;
  TD.IsolationLevel:= xilREADCOMMITTED ;
  Try
    SQLConnection1.StartTransaction( TD );
    Try
      // Bloco de Rotinas a serem executadas  durante a transação
      SQLConnection1.Commit( TD );
    Except
      On E: Exception Do
      Begin
        SQLConnection1.Rollback( TD );
        MessageDlg('Processamento não foi efetivado: ' + e.Message, mtError,
         [mbOK], 0 );
      End;
    End;
  Finally
    // liberação de todos os recursos alocados para o procedimento
  End;
end;
```

Listagem 4. Modelo de controle transacional com dbExpress 4

```
procedure TfoBaixaAutomaticaBoletos.ToolButton3Click(Sender: TObject);
var
 transacao : TDBXTransaction ;
begin
  transacao:=SQLConnection1.BeginTransaction(TDBXIsolations.Read
  Committed);
  try
    try
  // bloco de comandos da transação
  dmBase.SQLConnection1.CommitFreeAndNil( transacao ) ;
    Except
      on E: exception Do
      Begin
        dmBase.SQLConnection1.RollbackFreeAndNil( transacao );
        messageDlg('Processamento não foi efetivado: ' + e.Message, mtError,
         [mbOK], 0);
      End;
    end;
  finally
    FreeAndNil( transacao ) ;
  end;
end;
```

Nota

O método BeginTransaction é um método sobrecarregado, tendo duas assinaturas possíveis. Na primeira utiliza-se como parâmetro do método o tipo de isolamento desejado para a transação. A segunda assinatura é ainda mais simples, consiste simplesmente em uma chamada sem parâmetros. O método BeginTransaction(), criará uma transação com o nível de isolamento readcommited, considerado o nível de isolamento padrão. Estas duas chamadas têm o mesmo efeito:

```
transacao:=SQLConnection1.BeginTransaction
(TDBXIsolations.ReadCommitted);
transacao:=SQLConnection1.BeginTransaction();
```

Listagem 5. Código transacional para atualização de valores

```
01 procedure TForm1.SpeedButton1Click(Sender: TObject);
02 var
03   transacao : TDBXTransaction ;
04   sPorcentagem : String ;
05   fPorcentagem : double ;
06   query : TSQLQuery ;
07 begin
08   // validações de dados
09   if not sdsContasReceber.Active then
10     raise Exception.Create('Nenhuma pesquisa foi realizada');
11   if sdsContasReceber.IsEmpty then
12     raise Exception.Create('Nenhum registro foi encontrado');
13   if not InputQuery('Reajuste de valor',
14               'reajuste(%)', sPorcentagem) then
15     exit ;
16   if not TryStrToFloat(sPorcentagem, fPorcentagem )then
17     raise Exception.Create('Porcentagem informada não é válida');
18   if messageDlg('Confirma a atualização em ' +
19             FloatToStr(fPorcentagem) + ' %',
20             mtConfirmation, [mbOK, mbCancel], 0 ) <> mrOK then
21     raise Exception.Create('Atualização abortada pelo usuários');
22   query := TSQLQuery.Create( self );
23   query.SQLConnection := SQLConnection1 ;
24   transacao := SQLConnection1.BeginTransaction ;
25   try
26     try
27       query.Close ;
28       query.SQL.Text := ' UPDATE CONTASRECEBER SET VALOR = VALOR + ((
29                   VALOR * :pPorcentagem) / 100) ' +
30                  ' WHERE DATAVENCIMENTO BETWEEN :dtInicial AND
31                                   :dtFinal ' +
32                  ' AND (TOTALRECEBIDO IS NULL OR TOTALRECEBIDO
33                       = 0 ) ' ;
34       query.ParamByName('pPorcentagem').AsFloat := fPorcentagem ;
35       query.ParamByName('dtInicial').AsDate :=  dfdfsdsContasReceber.DataSet.
         ParamByName('dtInicial').AsDate ;
36       query.ParamByName('dtFinal').AsDate := sdsContasReceber.DataSet.
         ParamByName('dtFinal').AsDate ;
37       query.ExecSQL();
38       query.Close ;
39       query.SQL.Text := ' INSERT INTO HISTORICOOPERACAO ' +
40                  ' (DATA, DESCRICAO) ' +
41                  ' values (:DATA, :DESCRICAO) ' ;
42       query.ParamByName('Data').AsDate := date ;
43       query.ParamByName('descricao').AsString := 'Atualização de valor em ' +
         sPorcentagem + ' % ' ;
44       query.ExecSQL() ;
45       SQLConnection1.CommitFreeAndNil( transacao );
46     Except
47       on e : Exception do
48       begin
49         SQLConnection1.RollbackFreeAndNil( transacao );
50         MessageDlg('Erro durante o processamento: ' + e.Message, mtError, [mbOK], 0) ;
51       end;
52     end;
53   finally
54     query.Close;
55     FreeAndNil(query);
56     FreeAndNil(transacao);
57     sdsContasReceber.Refresh ;
58   end;
59 end;
```

executados dentro da mesma transação sem nenhum problema. Se tudo correr bem, o comando commit será executado e todas as alterações serão efetivadas na base de dados. Caso contrário, mesmo que apenas o último registro apresente algum tipo de problema, todas as alterações serão desfeitas pelo comando Rollback, sem afetar a integridade da base de dados. Para finalizar a procedure, garante-se a desalocação dos recursos utilizados dentro do bloco finally.

Como se pode perceber trabalhar com transações com dbExpress é uma tarefa bem simples, o importante a ser lembrado com relação às transações, independentemente do gerenciador de banco de dados que será acessado, é que as transações consomem muitos recursos do servidor e devem ser projetadas para que sua execução seja a mais rápida possível. Procure sempre coletar todos os dados antes de iniciar uma transação, evite ao máximo manter uma transação aberta aguardando interação com o usuário do sistema. Esse tipo de situação pode causar inúmeros problemas de performance nas aplicações.

Conclusão

A utilização de transações é um aliado muito importante para o desenvolvedor, principalmente em ambientes com muita concorrência a dados. A utilização desse recurso de forma consciente garantirá a integridade da base de dados e evita problemas de travamento de registros, que provocam muita lentidão e dores de cabeça. As transações devem sempre ser usadas em processamentos que devem garantir o total sucesso ou em caso de problema, recuperar o estado original dos dados antes do início da operação. Em processos que modificam diversas tabelas, a transação garante que as alterações serão completas, evitando que qualquer problema no decorrer da execução, modifique apenas parte da informação que deveria ser alterada ou inserida.

Jéfferson Ricardo Zuchi
jefferson@sygnux.com.br
Formado em processamento de dados pela FATEC-TQ. Atua como analista de sistemas Delphi na AMIL - Assistência Médica Internacional. Também atuou como professor na Faculdade de Tecnologia de Taquaritinga(FATEC-TQ) por 2 anos, lecionando disciplinas relacionadas a Delphi e Java.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Driver dbExpress para vários banco de dados
http://www.devart.com/dbx/

A Guide to Using dbExpress in Delphi database applications
http://delphi.about.com/od/database/a/dbexpressguide.htm

Migrating BDE Applications to dbExpress – John Kaster
http://edn.embarcadero.com/br/article/29106

Banco de dados relacional
http://pt.wikipedia.org/wiki/Banco_de_dados_relacional

DataSnap XE
Técnicas Avançadas

Como já exibido há tempos em edições anteriores da revista Clube Delphi, o DataSnap existe muito antes do Delphi XE aparecer. Em 1997 a Borland lançou o Delphi 3 e juntamente integrado a ele estava o MIDAS (Multitier Distributed Application Services), tecnologia que seguiu com esse nome até 2001, com o lançamento do Delphi 6, e foi renomeada para DataSnap. A biblioteca MIDAS muito antes do Delphi XE já era sinônimo de alto desempenho na criação de aplicações distribuídas que até antes do Delphi 2009 eram baseadas nas tecnologias DCOM, COM+ e BSS.

Vale lembrar que todas estas tecnologias derivaram do antigo COM, criado pela Microsoft em 1993 para suportar comunicação entre processos e muito utilizada em controles Active X, OLE, OLE Automation. O principal problema do DCOM (Distributed Component Object Model) foi a problemática da queda de performance conforme a conexão de clientes aumentava.

COM+ (versão melhorada do antigo Component Object Model) em outro caso, apresentava melhor performance em relação a este problema, porém, ainda utilizava o servidor de aplicação baseado em DLL, o que gerava um problema de distribuição e muitas vezes manutenção. Já o BSS (Borland Socket Server) era uma mistura de tudo isso, utilizando DCOM e Sockets, servidor de aplicação baseado em executável (e não mais DLL), porém, até este modelo que persistiu até a versão 2007 do Delphi, ainda havia a dependência com a tecnologia COM+.

A partir da versão do Delphi 2009 o DataSnap começou a ter uma evolução significativa. A dependência da tecnologia COM+ foi eliminada, e tanto o DataSnap quanto o Framework dbExpress sofreram grandes modificações, dando origem ao novo modelo de desenvolvimento multicamadas. O Delphi 2009 certamente foi um divisor de águas em relação à tecnologia multicamadas. A arquitetura de desenvolvimento de aplicações distribuídas possuía muito mais performance e mais simplicidade no desenvolvimento.

A versão 2009 marcou também o início do suporte à linguagem de notação JSON (Java Script Object Notation) que nada mais é que uma estrutura de dados Java Script muito leve e podendo ser dezenas de vezes mais rápida que a mesma representação de dados em XML. Vale lembrar que a maioria das linguagens já suporta JSON.

Na versão 2010 o mesmo suporte a JSON veio como o principal responsável pela volta do Callback, que não estava presente na versão anterior, iniciou-se também o suporte a servidores REST, transportando

Resumo DevMan

De que trata o artigo:

Utilização dos novos recursos encontrados no DataSnap XE como a transferência de objetos, autenticação de usuários, gerenciamento de sessões cliente/servidor, gerenciamento de conexões com qualquer banco de dados e controle de redundância.

Para que serve:

Utilizando a tecnologia DataSnap pode-se centralizar toda a regra de negócio dos nossos aplicativos em um único lugar. Após a implementação essas regras/métodos podem ser utilizadas em clientes tanto Web quanto Desktop independente da linguagem, utilizando protocolos comuns.

Em que situação o tema é útil:

A escalabilidade, segurança, e claro, a facilidade de manutenção são os principais pontos a serem considerados para quem se interessou pelo DataSnap. Escalabilidade devido ao servidor DataSnap poder ser consumido por qualquer tipo de cliente, segurança pois toda a implementação de métodos e controle de dados é feita em um único local e junto a isso a facilidade de manter um código centralizado.

DataSnap XE:

Geralmente quando está se iniciando em uma nova tecnologia é um pouco complicado utilizar todas as funcionalidades da mesma de maneira adequada nos primeiros projetos. Através deste artigo serão apresentadas de maneira simples algumas das diversas funcionalidades e recursos encontrados no DataSnap do RAD Studio XE, como a transferência de objetos JSON, uma forma de transporte muito eficiente, a autenticação de usuários em servidores com protocolo TCP/IP aumentando a segurança de seus servidores, um exemplo de como implementar Failover e fazer com que seu sistema trabalhe de forma mais inteligente sozinho, além é claro de otimizar a velocidade e o processamento de seu servidor, utilizando as conexões com o banco de dados de forma mais inteligente.

Alexandre José Gonçalves da Silva

dados com notação JSON nativamente. Um outro grande marco desta versão caracterizava a disponibilização do código-fonte da então "fórmula mágica", o MIDAS, que poderia ser facilmente melhorado. REST é a abreviação de Representational State Transfer ou Transferência de estado representacional. Ele surgiu da tese de doutorado sobre a Web, escrita por Roy Fielding, que foi um dos idealizadores do protocolo HTTP.

Na versão XE do RAD Studio foram realizadas mais melhorias consideráveis em relação às versões anteriores, lembrando que a compatibilidade entre as versões foi mantida. Na última versão foi possível trafegar, sem problemas, dados suportados tanto pelo dbExpress quanto dados JSON.

Para exemplificar melhor, todas as melhorias citadas até aqui, será exposto de maneira simples como utilizar um servidor DataSnap sem criar Transactional / Remote Data Modules (dependentes da tecnologia COM). Outra grande novidade, que facilmente é implementada, é o gerenciamento de conexões com o banco de dados, recurso comumente chamando de Pool de conexões e que embora poucos saibam, é suportado pelo dbExpress desde muito antes disso. Será apresentada a utilização de Generics para o manuseio deste Pool, a partir das conexões recebidas e gerenciadas pelo próprio servidor.

O transporte de objetos entre cliente e servidor utilizando classes genéricas, o que aumenta significativamente a possibilidade de utilização de POO entre as aplicações cliente e servidora, o que é demonstrado aqui por meio de uma classe de cadastro de CEPs.

Também é visto o conceito de redundância de servidores, que resolve um velho problema. Em todos os modelos de aplicação multicamadas (inclusive no DataSnap), quando a aplicação servidora fica indisponível (devido a problemas de conexão, por exemplo), a aplicação cliente torna-se inutilizável. Uma das grandes novidades das novas versões do DataSnap é justamente a facilidade de se implementar um gerenciador de servidores, transferindo a conexão com a aplicação servidora antiga para um novo servidor, que faz com que o problema se torne transparente.

Nota do DevMan

Para quem é desenvolvedor de aplicações multicamadas de longa data os recursos de Failover não são novidade, porém, antes eram implementados de forma muito mais trabalhosa. Este recurso permite que em caso de indisponibilidade de um servidor de aplicação outro assuma, tornando a aplicação cliente disponível novamente e evitando aborrecimentos. Este tipo de problema de indisponibilidade pode ocorrer de várias maneiras como quedas de conexão (perda do link de internet), problemas de rede ou infraestrutura, atualizações sem sucesso entre muitas outras. A solução de Failover permite que seja implementado não um, mas quantos servidores forem necessários para o caso de falha, bastando apenas redirecionar a conexão para a porta e host do outro servidor de aplicação ativo.

Já o conceito de Callback existe desde o começo do desenvolvimento de aplicações distribuídas. Trata-se de troca de mensagens entre aplicação servidora e cliente. Após a remoção do vínculo com a tecnologia COM na versão 2009, o novo modelo que surgia (DataSnap) acabou perdendo a funcionalidade de Callback, que após estar entre a lista de desejos para a nova versão, teve seu retorno na versão 2010, sendo realizado por meio da troca de mensagens via JSON.

Nota do DevMan

COM é a abreviação para Component Object Model que deu origem a COM + (uma versão melhorada). Consiste em uma plataforma da Microsoft para componentes de software lançada em 1993. Ela é usada para permitir a comunicação entre processos e a criação dinâmica de objetos em qualquer linguagem de programação que suportem a tecnologia. O termo COM é frequentemente usado no desenvolvimento de software para se referir a um grupo de tecnologias que incluem OLE, OLE Automation, ActiveX, COM+ e DCOM.

Foram tecnologias criadas pela Microsoft e muito tempo utilizadas. Foi amplamente utilizada para desenvolvimento de aplicações multicamadas de missão crítica e alta-disponibilidade e em outras tecnologias como o BSS (Borland Socket Server) até a versão do Delphi 2007. A partir da versão 2009 do Delphi, houve mudanças no desenvolvimento de aplicações multicamadas, aonde o problema de dependência do COM foi resolvida, surgindo então o modelo denominado de novo DataSnap que contava com a tecnologia dbExpress, que também sofreu grandes mudanças.

Nota

As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador, que é executado pelo compilador antes da execução do processo propriamente dito. Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Todas as diretivas do Delphi seguem o padrão de sintaxe {$DIRETIVA}. As diretivas podem ser colocadas em qualquer parte do programa. {$IFDEF VERXX} é um exemplo de diretiva que verifica a versão do compilador do Delphi onde a aplicação está sendo compilada.

Criando Server Methods em DataSnap

O DataSnap do RAD Studio possui alguns Wizards muito úteis, mas com o passar do tempo e com a própria experiência adquirida, pode-se facilmente criar em poucos minutos servidores de aplicação sem a utilização dos Wizards, eliminando do código desnecessário, dependendo da necessidade.

Listagem 1. Classe de métodos

```
unit ServerMethods;

interface

uses
  Classes, SysUtils;

type
{$METHODINFO ON}
 TServerMethods = class(TPersistent)
 public
   function Echo(const Value: string): string;
 end;
{$METHODINFO OFF}

implementation

{ TServerMethods }

function TServerMethods.Echo(const Value: string):
string;
begin
  Result := Value;
end;

end.
```

Listagem 2. Classe base para transporte JSON

```
type
 TJSONBaseObject = class
 private
 public
   class function BaseObjectToJSON<J: class>
   (destObject: J): TJSONValue;
   class function JSONToBaseObject<J: class>
   (JSON: TJSONValue): J;
 end;

implementation
...
```

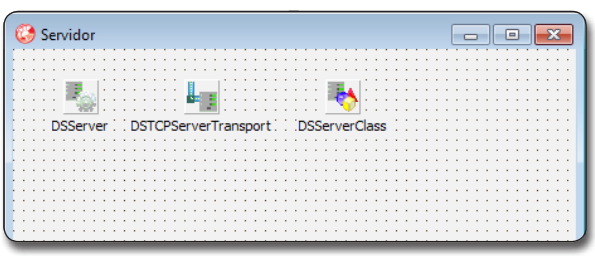


Figura 1. Estrutura inicial do servidor DataSnap

No RAD Studio XE pode-se desenvolver servidores DataSnap registrando uma classe qualquer para fornecer métodos remotamente. Para isso acontecer ela deve seguir duas regras: deve ser descendente de TPersistent e possuir a diretiva de compilação $MethodInfo ativada.

Na **Figura 1** é construída a aplicação servidora que contém os componentes TDSServer, TDSTCPServerTransport e TDSServerClass. Após a criação do modelo de servidor de aplicação, resta criar a classe responsável por disponibilizar os métodos do servidor de aplicação (**Listagem 1**).

Nessa classe é utilizada a diretiva de compilação {$METHODINFO ON} logo após a sua declaração. Isso permite registrar esta classe no componente TDSServerClass. Para isso, basta a implementação do evento OnGetClass, fazendo com que o parâmetro PersistentClass retorne a classe de métodos:

```
procedure TForm1.DSServerClassGetClass
(DSServerClass: TDSServerClass;
 var PersistentClass: TPersistentClass);
begin
 PersistentClass := TServerMethods;
end;
```

Desta maneira o servidor de aplicação já permite a disponibilização dos métodos da classe ServerMethods. No mesmo exemplo é criado o método Echo que apenas retorna a String recebida como parâmetro.

Transferência de objetos utilizando JSON

Como comentado anteriormente, o DataSnap foi reestruturado na versão 2009, mas não era possível transferir objetos entre o cliente e o servidor, apenas DatatTypes nativos do DBExpress.

Utilizando JSON pode-se facilmente trocar objetos entre clientes e servidores desenvolvidos em Delphi e outras linguagens. Para utilizar esse recurso primeiramente é criada uma classe que faça uso de Generics para a conversão de objetos (**Listagem 2**). Todas as demais classes que serão transportadas entre o cliente e o servidor deverão ser herdadas dela.

Ela contém no uses as units DBXJSON e DBXJSONReflect.

A classe base possui apenas dois métodos, BaseObjectToJSON, que é responsável por converter qualquer objeto descendente em JSON e JSONToBaseObject que faz justamente o contrário.

O método BaseObjectToJSON da **Listagem 3** recebe o objeto a ser convertido para JSON como parâmetro, checando se no mesmo existe conteúdo, caso não exista automaticamente é retornado um TJSONNull que seria um objeto nulo. Caso haja conteúdo no parâmetro é retornado um valor TJSONValue, obtido através de um TJSONMarshal que é liberado logo

Nota do DevMan

Uma constraint em Generics nada mais é do que uma restrição sobre os tipos reais que podem ser usados para parametrizar um tipo genérico. Ou seja, quando em tempo de execução é definido o tipo real com que se vai trabalhar, este tipo deve satisfazer a condição definida. Atualmente pode-se definir quatro tipos de restrições com suas variações:

Class – Define que o tipo real a ser utilizado deve obrigatoriamente ser um objeto descendente de TObject, exemplo de definição: myfunction<T: class>(Obj: T). Caso queira especificar que este objeto deve herdar de uma classe mais específica basta definir: myfunction <T: TCustomEdit>(Obj: T). Neste caso o objeto real passado deve ser obrigatoriamente descendente da classe TCustomEdit.

Interface – Define que o tipo real a ser utilizado deve obrigatoriamente descender de uma determinada interface, ou ser um objeto que implemente esta interface, por exemplo: myfunction<T: IInterfaceList>(Obj: T). Neste caso o objeto passado deve ser do tipo desta interface, descendente desta interface ou implementar esta interface.

Record – Define que o tipo real a ser utilizado deve ser obrigatoriamente uma estrutura de valor e não de referência, podendo ser um valor ordinário, ponto flutuante ou um Record, por exemplo: myfunction<T: record>(Obj:T).

Constructor – Define que o tipo real a ser utilizado deve possuir obrigatoriamente um construtor declarado que não possua parâmetros. Isso é muito comum quando internamente na função há a necessidade de se instanciar este tipo genérico. Assim há a certeza que pode-se invocar o método T.Create sem que hajam erros em tempo de execução. Um exemplo de definição seria: myfunction<T: constructor>(Obj:T).

Nota do DevMan

A partir da versão 2009 do Delphi a instrução Exit passou a considerar retornos em sua chamada semelhante ao C++ e ao Delphi Prism, ou seja, não é necessário especificar um Result para a saída da função antes de executar um Exit.

Listagem 3. Método para converter de Object para JSON

```
class function TJSONBaseObject.BaseObjectToJSON<J>(destObject: J): TJSONValue;
var
 FJSON: TJSONMarshal;
begin
 if Assigned(destObject) then
 begin
  FJSON := TJSONMarshal.Create(TJSONConverter.Create);
  try
   Exit(FJSON.Marshal(destObject));
  finally
   FJSON.Free;
  end;
 end else
  Exit(TJSONNull.Create);
end;
```

Listagem 4. Método para converter de JSON para Object

```
class function TJSONBaseObject.JSONToBaseObject<J>(JSON: TJSONValue): J;
var
 FJSON: TJSONUnMarshal;
begin
 if not (JSON is TJSONNull) then
 begin
  FJSON := TJSONUnMarshal.Create;
  try
   Exit(J(FJSON.Unmarshal(JSON)));
  finally
   FJSON.Free;
  end;
 end else
  Exit(nil);
end;
```

Listagem 5. Server Methods

```
type
{$METHODINFO ON}
 TServerMethods = class(TPersistent)
 public
  function Echo(const Value: string): string;
  function GetJSONCEP(const CEP: Integer): TJSONValue;
 end;
{$METHODINFO OFF}

implementation

{ TServerMethods }

function TServerMethods.Echo(const Value: string): string;
begin
 Result := Value;
end;
function TServerMethods.GetJSONCEP(const CEP: Integer): TJSONValue;
var
 FJSONCEP : TJSONCEP;
begin
 FJSONCEP := TJSONCEP.Create;
 try
  FJSONCEP.Bairro      := 'Bairro pesquisado';
  FJSONCEP.CEP         := CEP;
  FJSONCEP.Cidade      := 'Cidade pesquisada';
  FJSONCEP.Estado      := 'Estado pesquisado';
  FJSONCEP.Logradouro := 'Logradouro pesquisado';
  Result := FJSONCEP.BaseObjectToJSON<TJSONCEP>(FJSONCEP);
 finally
  FJSONCEP.Free;
 end;
end;
```

em seguida. No código da **Listagem 4** é importante atentar que a primeira validação verifica se o parâmetro passado é um TJSONNull (conforme a **Listagem 3**), sendo um JSON com conteúdo válido, o objeto TJSONUnMarshal é criado para retornar o objeto original, ou seja, o TJSONUnMarshal desfaz o processo de conversão do TJSONMarshal da **Listagem 3**.

Uma vez terminada a classe TJSONBaseObject, já é possível transferir objetos inteiros entre cliente e servidor. Para exemplificar os métodos criados, é proposta uma nova classe chamada TJSONCep que herda de TJSONBaseObject e que gerencia as informações de endereço no padrão estabelecido pelos Correios, contendo as seguintes propriedades: Bairro, CEP, Cidade, Estado, Logradouro.

A classe TJSONCep não possui nenhuma particularidade específica, o único intuito é transportar informações referentes ao endereço solicitado. Na **Listagem 5** a unit ServerMethods criada para disponibilizar os métodos do servidor de aplicação para a aplicação cliente é alterada, tendo adicionado o método GetJSONCEP que retornará um endereço a partir de um CEP

Nota do DevMan

ProxyClient é uma classe gerada automaticamente a partir do DBX Framework, aonde são disponibilizados os métodos criados na aplicação servidora para a chamada na aplicação cliente. Desta forma, cada ServerModule funciona como uma instância de uma classe com seus respectivos métodos. A grande vantagem é a possibilidade de criar um objeto e executar os métodos remotos de forma como se fossem locais.

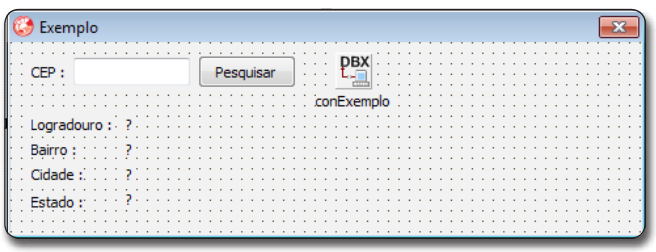


Figura 2. Layout da aplicação cliente

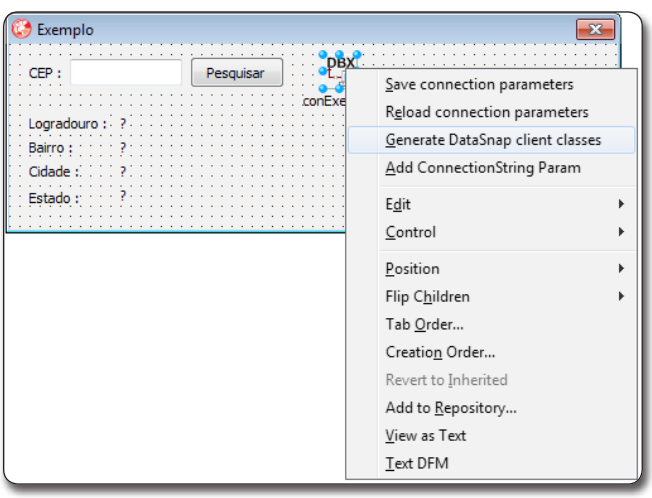


Figura 3. Gerando a Proxy Client

informado. No exemplo, os valores são pré-definidos para os atributos criados na classe JSONCEP, porém, nada impede de consultar uma base de dados para retornar estes valores à aplicação cliente.

A seguir é criada a aplicação cliente, o layout proposto encontra-se na **Figura 2**. Sendo assim, é feita a chamada do método na aplicação cliente, porém, para acessar os métodos do servidor de aplicação é necessário fazer a chamada da ProxyClient através do SQLConnection. Com o servidor ativo basta gerar a classe ProxyClient (**Figura 3**). Na **Listagem 6** são ilustrados

Figura 4. Aplicação cliente para teste de conexão

Listagem 6. Recebendo a informação do server

```
...
var
  frmExemplo: TfrmExemplo;
  ServerMethods: TServerMethodsClient;

implementation

uses
  JSONCEP;

{$R *.dfm}

procedure TfrmExemplo.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  FreeAndNil(ServerMethods);
end;

procedure TfrmExemplo.FormCreate(Sender: TObject);
begin
  conExemplo.Connected := True;
  ServerMethods := TServerMethodsClient.Create(conExemplo.DBXConnection);
end;

procedure TfrmExemplo.btnCEPClick(Sender: TObject);
var
  FJSONCEP: TJSONCEP;
begin
  FJSONCEP := TJSONCEP.JSONToBaseObject<TJSONCEP>(ServerMethods.
  GetJSONCEP(StrToInt(edtCEP.Text)));
  try
    lblBairro.Caption := FJSONCEP.Bairro;
    lblCidade.Caption := FJSONCEP.Cidade;
    lblEstado.Caption := FJSONCEP.Estado;
    lblLog.Caption    := FJSONCEP.Logradouro;
  finally
    FJSONCEP.Free;
  end;
end;
```

os eventos OnCreate, responsável por abrir a conexão e instanciar a classe de ServerMethods da ProxyClient, OnClose, que faz a destruição e liberação da classe e a chamada ao método do servidor através no evento OnClick do botão de pesquisa.

Estão sendo utilizadas as mesmas classes do servidor de aplicação, a diferença é que aqui está sendo feito o caminho inverso, ou seja, JSONToBaseObject. Lembrando que o objeto TJSONCEP é criado no servidor, transportado para o cliente e destruído nele após a utilização. Dessa maneira é possível perceber como o DataSnap simplifica e centraliza as operações. Como já dito, facilmente é possível integrar consultas com banco de dados e validar informações no servidor, transportando para o cliente apenas o resultado relevante de maneira muito leve e rápida.

Autenticação de usuários com DataSnap

Um recurso muito importante que é possível implementar em um servidor DataSnap é a autenticação de usuários. Através dela pode-se bloquear o acesso dos métodos remotos de qualquer desconhecido.

Por exemplo: em um cenário onde se tem um servidor de métodos recebendo várias requisições durante o dia. Como é possível garantir que apenas usuários habilitados possam se conectar no servidor? Supondo que apenas usuários cadastrados no banco de dados pudessem se conectar. Este cenário é reproduzido a partir de um exemplo bem simples e prático de como proteger o servidor de usuários não autorizados.

O objetivo é implementar autenticação baseada em usuários existentes num banco de dados e criar um log simples de tentativas de conexão malsucedidas para monitoramento. Para isso, é criado um banco de dados simples em Firebird que contém duas tabelas: USERS(LOGIN: INTEGER, PASS:VARCHAR(25) e LOG_TENTATIVA_ACESSO(ID: INTEGER, LOGIN:VARCHAR(25), DATA: TIMESTAMP). Dentro da tabela USERS são adicionados dois registros conforme mostra o script a seguir:

```
INSERT INTO USERS (LOGIN, PASS) VALUES ('delphi', 'devmedia');
INSERT INTO USERS (LOGIN, PASS) VALUES ('devmedia', 'delphi');
```

Ainda aproveitando o servidor de aplicação já criado, as implementações necessárias são feitas nele, iniciando pela adição de um SQLConnection com a conexão previamente configurada. A **Listagem 7** ilustra o evento OnConnect do componente TDSServer.

O que foi feito é semelhante à tradicional validação que já é feita em telas de login de usuário. Utilizando o evento OnConnect do componente DSServer tem-se o parâmetro DSConnectEventObject. Através deste parâmetro é possível obter informações referentes à conexão, que é exatamente o que é feito. Para a consulta do usuário logado no banco de dados foi criado um SQLQuery que recebe a instrução SQL para busca de informações. A grande diferença encontra-se nas linhas 9, 10, 17 e 18 respectivamente, pois, ao invés de criar um método de autenticação e passar os parâmetros para o mesmo, o método OnConnect poupa o trabalho já fazendo este serviço. Nas linhas mencionadas são utilizados os próprios valores passados da conexão de aplicação cliente através da linha:

```
DSSConnectEventObject.ConnectProperties.
Values['nome_do_parametro'];
```

Também é interessante mencionar que caso a Query se encontre vazia, será efetuada uma inserção dentro da tabela de log com as informações do login que tentou acessar o servidor, e consequentemente gerando uma exception para a aplicação

Listagem 7. Configurando a autenticação

```
01 procedure TForm1.DSServerConnect(DSConnectEventObject:
     TDSConnectEventObject);
02 var
03   FQuery: TSQLQuery;
04 begin
05   FQuery := TSQLQuery.Create(nil);
06   try
07     FQuery.SQLConnection := conServidor;
08     FQuery.SQL.Text   :=
       'select 1 from users where login = :login and pass = :pass';
09     FQuery.Params.ParamByName('login').Value :=
       DSConnectEventObject.ConnectProperties.Values['DSAuthenticationUser'];
10     FQuery.Params.ParamByName('pass').Value :=
       DSConnectEventObject.ConnectProperties.Values['DSAuthenticationPassword'];
11     try
12       FQuery.Open;
13       if FQuery.IsEmpty then
14       begin
15         FQuery.Close;
16         FQuery.SQL.Text :=
           'insert into log_tentativa_acesso(login, data) values (:login, :data)';
17         FQuery.Params.ParamByName('login').Value :=
           DSConnectEventObject.ConnectProperties.Values['DSAuthenticationUser'];
18         FQuery.ParaMs.ParamByName('data').Value := Now;
19         FQuery.ExecSQL;
20         raise Exception.Create('Conexão negada pelo servidor. Usuário
           e/ou senha inválidos');
21       end
22       else
23         FQuery.Close;
24     except
25       on e: Exception do
26       begin
27         raise Exception.Create(e.Message);
28       end;
29     end;
30   finally
31     FQuery.Free;
32   end;
33 end;
```

cliente. A seguir é mostrada a aplicação cliente para efetuar os testes de conexão (**Figura 4**) que implementa os eventos OnClick do botão e AfterConnect do SQLConnection (**Listagem 8**).

O evento OnClick recebe nos parâmetros de conexão (DSAuthenticationPassword e DSAuthenticationUser) os dados referentes ao login e senha. Caso a conexão seja bem sucedida o evento AfterConnect é disparado, exibindo uma mensagem de erro, caso contrário, é exibida a mensagem de erro, disparada pelo servidor de aplicação.

Listagem 8. Eventos OnClick e AfterConnect

```
procedure TfrmExemplo.btnConectarClick(Sender: TObject);
begin
  conExemplo.Close;
  conExemplo.Params.Clear;
  conExemplo.Params.Append('drivername=DATASNAP');
  conExemplo.Params.Append('CommunicationProtocol=tcp/ip');
  conExemplo.Params.Append('DSAuthenticationPassword=' + edtPass.Text);
  conExemplo.Params.Append('DSAuthenticationUser=' + edtUser.Text);
  conExemplo.Connected := True;
end;

procedure TfrmExemplo.conExemploAfterConnect(Sender: TObject);
begin
    Application.MessageBox('Conectado com sucesso!',
      PChar(Application.Title), MB_OK + MB_ICONINFORMATION);
end;
```

Gerenciamento de conexões com o banco de dados através de sessões

Um recurso muito importante e útil introduzido no RAD Studio XE é o gerenciamento de sessões de maneira mais facilitada. Em particular isso pode ajudar em várias tarefas, e uma delas é o gerenciamento de conexões de banco de dados.

Muitos desenvolvedores utilizam uma conexão com o banco de dados nativa dentro de cada Server Module, o que pode ocasionar entre outras coisas uma baixa performance do banco de dados devido à falta de memória e o próprio travamento do servidor de aplicação. Isto porque, quando um cliente se conecta ao servidor e invoca um método remoto, uma conexão com o banco de dados é iniciada e só finalizada quando o cliente se desconecta. Agora basta imaginar um servidor com pelo menos 5 Server Modules diferentes sendo acessados por 20 clientes distintos. Nesse exemplo simples já há um acúmulo de 100 conexões com o banco de dados abertas e dependendo das configurações de memória por conexão, ocasionando lentidão de diversos processos.

Desenvolvendo de maneira racional todos os Server Modules do sistema deveriam se conectar a uma única conexão com o banco de dados que é relacionada ao cliente conectado. Deste modo, pode-se resolver ou otimizar este problema implementando um simples gerenciamento de conexões. O primeiro passo para isso é a declaração de uma variável do tipo TDictionary, presente em Generics.Collections e que armazenará as conexões e, método que retornará a conexão:

```
...
 private
   FConnection : TDictionary<Integer, TSQLConnection>;
   { Private declarations }
 public
   { Public declarations }
   function GetConnection : TSQLConnection;
...
```

No evento OnCreate é instanciado o dicionário que receberá as conexões:

```
procedure TfrmServidor.FormCreate(Sender: TObject);
begin
  //Adicione Generics.Collections ao uses
  FConnection := TDictionary<Integer, TSQLConnection>.
  Create;
end;
```

A **Listagem 9** ilustra o método GetConnection. Na linha 06 é utilizado o TDSSessionManager.GetThreadSession.ID que é o identificador único da sessão que está se conectando. Este ID é pesquisado dentro do Pool (FConnection) através do método ContainsKey, retornando a conexão encontrada caso exista. Caso não seja

encontrada, uma nova conexão é criada, adicionada ao Pool (linha 14) e retornada a quem requisitou. A conexão só é encerrada quando o cliente se desconecta da sessão (evento OnDisconnect do DSServer) e o Pool liberado da memória no momento em que o servidor é finalizado.

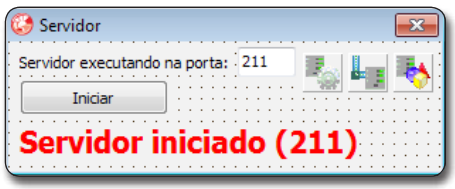


Figura 5. Servidor DataSnap modificado

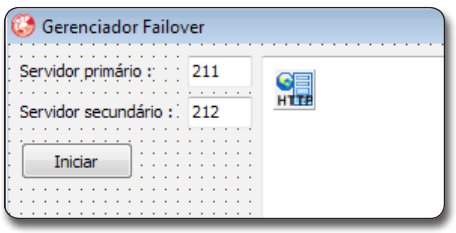


Figura 6. Layout do servidor Failover

Nota

Devido à extensão do aplicativo apenas a parte principal do código do gerenciador será apresentada no artigo, porém, o código-fonte completo encontra-se no site da DevMedia.

No método OnConnect do DSServer exibido na **Listagem 7**, o objeto FQuery pode ser alterado para obter a conexão a partir do Pool de conexões criado, como a seguir:

```
...
FQuery.SQLConnection := frmServidor.GetConnection;
...
```

Seguindo a linha de raciocínio do Pool da **Listagem 9**, caso a aplicação cliente execute o método PesquisarCliente primeiramente, o método GetConnection cria uma nova conexão e a adiciona ao dicionário (Pool) e, ao ser executado o método PesquisarFornecedor, ele apenas devolve a conexão já existente. Essa forma de trabalho é uma das mais importantes implementações a ser feita em servidores DataSnap tratando-se de otimização de servidores.

Redundância de servidores (failover)

No RAD Studio XE implementar Failover de servidores se tornou uma tarefa muito simples. Introduzido no Delphi 2010, o HTTP Tunneling permite que o fluxo de informações seja controlado entre o cliente e o servidor além de direcionar para onde for necessário.

Para implementar Failover é necessário desenvolver um gerenciador, responsável por gerenciar todos os dados solicitados/recebidos. O processo de desenvolvimento é basicamente o mesmo dos anteriores. Aqui é adaptado o servidor de aplicação (**Figura 5**) e implementada a codificação da **Listagem 10** que apenas configura o servidor para operar na porta informada no Edit.

A função Echo do Server Methods também é adaptada para retornar a porta em que a chamada do servidor é realizada. Para finalizar é desenvolvido um gerenciador de Failover para o redirecionamento da porta para outros servidores, que funciona como um Middleware, e a aplicação cliente de testes. A **Figura 6** ilustra o layout do aplicativo proposto. A **Listagem 11** demonstra a principal rotina do gerenciador de Failover.

Nela é criado um DBXDataSnapProperties que armazena os dados de conexão do

Listagem 9. Método GetConnection, responsável por retornar a conexão

```
01 function TfrmServidor.GetConnection: TSQLConnection;
02 var
03   SQLConnection : TSQLConnection;
04 Begin
05   //Adicione DSService ao uses
06   if FConnection.ContainsKey(TDSSessionManager.GetThreadSession.Id) then
07     Result := FConnection[TDSSessionManager.GetThreadSession.Id]
08   else
09   begin
10     SQLConnection := TSQLConnection.Create(nil);
11     SQLConnection.Params.Clear;
12     SQLConnection.LoadParamsOnConnect := true;
13     SQLConnection.ConnectionName := 'CD128';
14     FConnection.Add(TDSSessionManager.GetThreadSession.Id,
       SQLConnection);
15     Result := SQLConnection;
16   end;
17 end;
```

Listagem 10. Implementação do TButton

```
procedure TfrmServidor.btnIniciarClick(Sender: TObject);
begin
  try
    dstServidor.Port := StrToInt(edtPorta.Text);
  except
    dstServidor.Port := 211;
  end;
  dssServidor.Start;
  lblPorta.Caption := 'Servidor iniciado (' + edtPorta.Text + ')';
  lblPorta.Visible := True;
end;
```

Listagem 11. Implementação do gerenciador de failover

```
procedure TfrmGerenciador.Call(Sender: TObject; Session: TDSTunnelSession;
  Content: TBytes; var Count: Integer);
var
  FDBX: TDBXDatasnapProperties;
begin
  if Sender is Exception then
    mmoLog.Lines.Append(Exception(Sender).Message);
    FDBX := TDBXDatasnapProperties.Create(nil);
    try
    if Session.UserFlag = 1 then
      raise Exception.Create('Servidor secundário está off-line.');
      FDBX.Values[TDBXPropertyNames.DriverName] := 'Datasnap';
      FDBX.Values[TDBXPropertyNames.HostName] := 'localhost';
      FDBX.Values[TDBXPropertyNames.Port] := edt2.Text;
      try
      Session.Reopen(FDBX);
      Session.UserFlag := 1;
    except
      raise Exception.Create('Erro ao abrir servidor de backup.');
    end;
  finally
    FDBX.Free;
  end;
end;
```

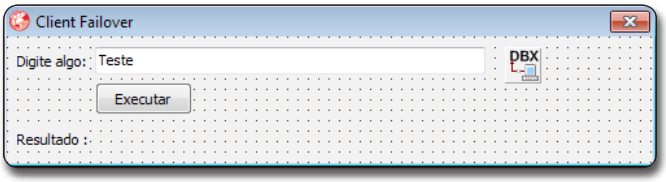


Figura 7. Cliente para testes de failover

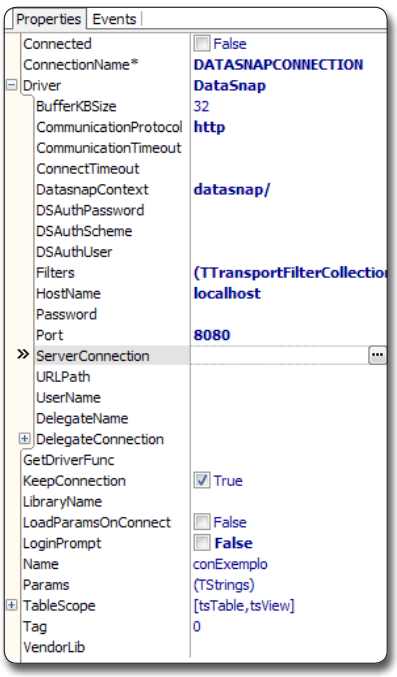


Figura 8. Propriedades de conexão no cliente

servidor secundário e os repassa à sessão de controle. Caso o servidor secundário já esteja em operação e esteja desabilitado, o gerenciador de Failover emite um erro, avisando sobre a não operabilidade do sistema. Nesse ponto pode-se implementar facilmente um sistema de rodízio de servidores, onde o gerenciador sempre tenta conectar no primeiro servidor DataSnap disponível. A aplicação cliente de testes encontra-se na **Figura 7**, bem como suas configurações de conexão (**Figura 8**).

Na **Figura 8** é definida a conexão DataSnap do cliente para a porta 8080 e o protocolo para http. Feito isso, basta conectar ao gerenciador de Failover e gerar novamente as classes ProxyClient. Na execução da aplicação cliente é possível notar a comunicação entre os servidores escolhidos.

Conclusão

O DataSnap sem dúvidas chegou para ficar. Existem vários outros recursos que já foram apresentados em outros artigos, como e principalmente o REST, protocolo adicionado no RAD Studio XE e que permite a comunicação entre servidores Delphi com outras linguagens. Deve-se lembrar também que é de extrema importância analisar todos os recursos que o Framework proporciona e saber adequá-lo da melhor maneira possível às nossas necessidades.

Alexandre José Gonçalves da Silva
alexandrej@live.com
Blog: www.alexandrejose.blog.br
Twitter: @alexandrejose
Analista de Sistemas graduado em Sistemas de Informação e Pós-graduado em Engenharia de Software pela PUCPR. Trabalha com diversas tecnologias e linguagens de programação, sendo Delphi a principal e desde 1999. Membro atuante em diversas comunidades e sempre defendendo com muito entusiasmo e paixão o nosso Delphi.

O que é JSON, como e quando utilizar?
http://www.tidbits.com.br/o-que-e-json-como-e-quando-utilizar

DataSnap
http://en.wikipedia.org/wiki/DataSnap

Welcome to the DataSnap area!
http://edn.embarcadero.com/br/delphi/distcomp/datasnap

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Padrões Criacionais
Fabrique famílias de objetos

Os padrões GoF documentam cinco padrões criacionais: Builder, Abstract Factory, Factory Method, Prototype e Singleton. Apesar de estes padrões possuírem implementações diferentes entre si, resumidamente todos têm o objetivo de controlar como as classes do projeto serão criadas, evitando o uso descontrolado de instruções como "TMinhaClasse.Create". A diferença entre estes padrões está nas consequências que cada um deles carrega consigo – consequências boas e/ou ruins.

O problema em espalhar o uso do construtor das classes sem controle algum é que o projeto fica preso às implementações quando deveria estar preso às interfaces. Consequentemente um projeto assim torna-se inflexível, o que aumenta o tempo gasto com manutenções. Imagine um projeto com uma interface ICarro. Esta interface possui diversas implementações diferentes, como por exemplo, TMustang, TFerrari, TAudiR8 e assim por diante. Se a criação/fabricação destas classes não for separada e controlada em um único local, em pouco tempo o projeto estará confuso e cheio desvios (instruções "if else", por exemplo), necessários para decidir qual das implementações de ICarro deve ser usada.

Neste cenário adicionar um novo tipo de carro, será uma tarefa cansativa e talvez desastrosa, pois o programador terá que procurar e alterar todos os locais onde existe uma tomada de decisão para descobrir qual tipo de carro deve ser criado. Este tipo de ação normalmente gera erros, exatamente por ser uma tarefa cansativa. A situação citada neste exemplo pode ser facilmente contornada se a forma comum de se criar um objeto, usando "TMinhaClasse.Create" for evitada.

Resumo DevMan

De que trata o artigo:

O artigo aborda quatro dos cinco padrões GoF criacionais (Builder, Abstract Factory, Factory Method, Prototype e Singleton). Neste artigo você aprenderá como implementar os padrões Abstract Factory, Factory Method, Prototype e Singleton.

Para que serve:

Todo projeto precisa criar instâncias de classes, mas se isto não for feito de forma controlada, o projeto tende a tornar-se complexo e rígido, dificultando a manutenção do mesmo. Os padrões criacionais são técnicas que ajudam controlar como as instâncias de classes são criadas.

Em que situação o tema é útil:

Todo projeto de software orientado a objetos precisa criar instâncias de suas classes. Os padrões criacionais são técnicas que tornam o sistema menos rígido permitindo que as instâncias sejam criadas sem que instruções como TMinhaClasse.Create sejam espalhas descontroladamente.

Padrões criacionais:

Neste artigo sobre os padrões de criação, serão abordados quatro dos cinco padrões GoF Criacionais. Você aprenderá como construir verdadeiras fábricas de objetos, capazes de controlar a criação de famílias inteiras de classes usando os padrões Abstract Factory, Factory Method, Prototype e Singleton.

O problema

Dando continuidade ao exemplo dos carros, imagine um projeto para jogo de corridas. Neste projeto existem diversos tipos de carros: TMustang, TFerrari, TAudiR8 e assim por diante. Essas classes são usadas em diversos pontos do projeto. Para poder criar um tipo de carro, é sempre necessário avaliar algum parâmetro para decidir qual o carro escolhido pelo usuário. Sendo assim, sempre que um carro precisa ser criado, uma instrução parecida com a **Listagem 1** deve ser escrita.

O código é extremamente simples. Talvez seja devido a esta simplicidade que este tipo de codificação não receba a devida atenção. A verdade é que apesar de simples, instruções como essa podem causar verdadeiros desastres ao projeto. Se um carro precisar ser criado em vários pontos diferentes do projeto, logo instruções como a da listagem em questão estarão espalhadas por todo o projeto e quando um novo tipo de carro for adicionado, todos estes pontos deverão ser alterados.

Esta forma de programar – espalhando a criação dos carros em vários pontos – desrespeita dois princípios importantes: Programe Para Interface e o Princípio de Responsabilidade Única. A seguir veja um pouco mais sobre estes dois princípios.

Princípios de orientação a objetos

Programar orientado a objeto é tão simples quanto programar de forma estruturada, ainda mais quando se programa em Object Pascal usando Delphi como IDE.

Nota

Você não pode conectar o plugue na tomada, pois o plugue possui uma interface diferente da tomada disponível (**Figura 1**).

RAFAEL STAVARENGO

Para criar um projeto orientado a objetos, a única coisa que o programador precisa fazer é criar classes: ele não precisa implementar padrões ou seguir princípios. Usando o Delphi, fica ainda mais fácil, pois o programador estará criando um projeto orientado a objetos mesmo que não saiba o que isto significa.

Apesar de ser simples se o programador não seguir alguns princípios e padrões quando criar um projeto orientado a objetos, ele provavelmente terá muitos problemas, devido à falta de organização estrutural do código. Isto significa que a POO só é vantajosa quando usada de maneira apropriada: usando os padrões e princípios comprovados pela comunidade de desenvolvedores.

Estes padrões e princípios são técnicas que ajudam o desenvolvedor criar um projeto que aproveite o máximo das vantagens que a POO oferece, como: flexibilidade, simplicidade e baixo custo de manutenção. De fato, os padrões de projetos são técnicas que usam os princípios da OO para solucionar problemas de projetos. Por serem guiadas pelos princípios as soluções propostas pelos padrões de projetos acabam sendo flexíveis e reutilizáveis.

Nota do DevMan

No dicionário Michaelis a palavra Polimorfismo é descrita como "algo que assume ou passa por várias formas". É exatamente isto. Imagine que você tenha a classe TPato, e diversas outras classes filhas: TPatoReal, TPatoBorracha, TPatoMadeira. Quando você cria um método que recebe um TPato como parâmetro (procedure Voar(Pato: TPato)), poderá na verdade passar qualquer objeto que seja subclasse de TPato. Isto é o polimorfismo, porque você tem um método que recebe um TPato que voará, porém de acordo com o tipo de pato que você passar o resultado do voo será diferente, ou seja, várias formas diferentes (polimorfo), de voar sem mudar uma linha do código – o único que realmente voa é TPatoReal.

Programe para interfaces

"Interface" no contexto do princípio "programe para interfaces" não tem que ser necessariamente uma entidade do tipo interface. Este princípio se refere ao uso do polimorfismo, que também é conseguido com uma classe abstrata.

Os projetos de software, na maioria das vezes, são constituídos por várias tarefas que atingem o mesmo objetivo traçando caminhos diferentes, por exemplo, uma aplicação pode exportar os dados bancários para mais de um banco diferente, entre outras coisas. Em um projeto orientando a objetos, cada uma destas tarefas será encapsulada em uma classe, o que significa que existirão grupos de classes parecidas que atingem o mesmo objetivo passando por caminhos diferentes.

Estes grupos de classes normalmente têm os mesmos métodos e atributos, visto que atingem o mesmo objetivo. Por exemplo, poderia existir um grupo de classes especializadas em exportar os boletos da empresa para o Banco Itaú, Banco do Brasil e Caixa Econômica Federal. Este grupo seria composto pelas classes TExportacaoBancoItau, TExportacaoBancoBrasil e TExportacaoCaixaEconomica. Uma vez que estas classes têm o mesmo objetivo (exportar boletos), elas também teriam os mesmos métodos.

Para formalizar que estas classes fazem parte do mesmo grupo é necessário criar "um contrato" que determine quais os métodos necessários para o grupo. Assim qualquer classe poderá fazer parte deste grupo se implementar os métodos declarados no contrato. Em outras palavras, o contrato determina a interface do grupo bem como determina os métodos que o grupo disponibiliza para o restante da aplicação. O contrato pode ser criado com uma classe abstrata ou interface, dependendo da situação. O importante é que o contrato determina como as outras partes do projeto interagem com um determinado grupo.

Listagem 1. Criando um tipo de carro

```
01  Var
02    Carro: ICarro;
03  Begin
04    if GetTipo = 1 then
05      Carro := TMustang.Create
06    else if GetTipo = 2 then
07      Carro := TFerrari.Create
08    else if GetTipo = 3 then
09      Carro := TAudiR8.Create
10    Else
11      raise Exception.Create('Tipo de carro desconhecido');
12    //Restante do código
13  end;
```

Figura 1. Interfaces diferentes

Apenas criar a interface para os grupos de classes especialistas, seja com classe abstrata ou com interfaces, não significa respeitar o princípio programe para interface. Para respeitar este princípio é necessário que o projeto não dependa das classes que implementam a interface, ou seja, o projeto não pode depender das classes que fazem parte do grupo. Para o exemplo das classes de exportação, isto significa que as demais partes do projeto usam as classes TExportacaoBancoItau, TExportacaoBancoBrasil e TExportacaoCaixaEconomica, sem depender delas, sem saber que elas existem. Fazendo isto, o princípio programe para interfaces começa a ser respeitado.

A grande dificuldade em conseguir usar o objeto de uma classe sem depender dela, é que não existe outra maneira de criar um objeto, se não invocando o construtor da classe. Você consegue imaginar uma forma de criar a instância de uma classe sem usar a instrução "TMinhaClasse.Create"? Até o final deste artigo você aprenderá como partes diferentes do projeto podem criar um objeto sem usar esta instrução e poderá respeitar o principio programe para interfaces (se preferir).

Princípio de responsabilidade única

Responda rápido: qual é a constante presente em todos os projetos de software independente da linguagem?

Um projeto sempre muda, seja para adicionar um novo comportamento, melhorar algo existente ou corrigir algum bug. Portanto para se manter à frente das necessidades do cliente, as alterações devem estar prontas tão rápidas quanto chegam os novos requisitos. O caminho mais fácil para conseguir isto é criar um projeto com classes coesivas, que se especializam em executar apenas uma tarefa da melhor maneira possível.

A coesão mede o quanto uma classe é específica para uma determinada tarefa, sendo que quanto mais coesiva melhor. Uma classe com alta coesão tende a ser alterada poucas vezes já que não influencia e não é influenciada por outras partes do projeto, ou seja, ela pode ser alterada sem influenciar partes do projeto que estão funcionando corretamente. Além disto, as classes coesivas dificilmente precisam ser alteradas quando outras partes do projeto são modificadas, pois elas realizam suas tarefas sem depender do restante do projeto.

O princípio de responsabilidade única é respeitado quando uma classe é coesiva, já que este princípio defende que toda classe deve ter apenas uma única responsabilidade (uma única razão para mudar). Por exemplo, TExportacaoBancoBrasil deve saber gerar o conteúdo para o arquivo de exportação, mas não deve saber como e onde salvá-lo. As responsabilidades de criar e salvar um arquivo devem pertencer à outra classe. Para saber se uma classe possui mais de uma responsabilidade, basta perguntar a você mesmo: quantas razões existem para mudar esta classe? Cada razão para mudar é considerada uma responsabilidade.

Por conta disto, o princípio de responsabilidade única muitas vezes é descrito como aquele que defende que uma classe deve ter apenas um único motivo para mudar. Voltando ao exemplo das exportações bancárias, se TExportacaoBancoBrasil souber onde e como salvar o conteúdo da exportação ela passa a ter mais de um motivo para mudar: ela muda quando o layout de exportação for atualizado bem como quando a forma ou local de salvar o conteúdo mudar. Neste caso, para respeitar o princípio de responsabilidade única, é necessário colocar uma destas duas funções (gerar conteúdo/salvar conteúdo) em uma nova classe.

Entre gerar o conteúdo e salvar o conteúdo de exportação, TExportacaoBancoBrasil está mais para "gerar conteúdo", pois este é o motivo dela existir. Sendo assim, a responsabilidade de "salvar conteúdo" está sobrando e deve ser separada em outra classe.

Respeitar os princípios é algo essencial para o amadurecimento de um projeto de software e o caminho mais fácil para respeitá-los é conhecer e aplicar os padrões de projetos. A seguir você será apresentado aos padrões criacionais, que são uma boa alternativa para repetir o princípio de responsabilidade única e o princípio programe para interface.

Factory Method

Factory Method é um padrão usado para permitir que subclasses decidam qual implementação a superclasse deve usar ou para disponibilizar um ponto comum de criar objetos de uma família de classes. O primeiro caso de uso é menos comum e normalmente é usado em conjunto com o padrão Template Factory. Uma das vantagens desta abordagem é que a superclasse fica livre de instruções de desvio como if.. else. Além disto, delegar a criação de objetos para as subclasses torna a superclasse mais dinâmica.

Usar o padrão Factory Method para disponibilizar um ponto comum de criação de objetos de um mesmo grupo de classes é a abordagem mais popular deste padrão. Esta técnica define que um método deve ser declarado na super classe para fabricar instâncias das subclasses de acordo com algum parâmetro. Este método considerado método de fábrica, normalmente é estático, isto significa que ele pode ser invocado diretamente de uma classe abstrata. Métodos estáticos são métodos declarados em nível de classe, por isto não é necessário criar objetos para invocá-los. A **Listagem 2** mostra a diferença entre um método estático e um método comum.

Para declarar um método estático é necessário adicionar a palavra chave class antes da declaração, seja este método uma procedure ou uma function (linhas 4 e 5). A principal diferença entre estes métodos torna-se evidente ao usá-los. Enquanto o método comum precisa de um objeto para ser usado (linha 19 e 20), o método estático pode ser invocado diretamente da classe (linha 18). A principal vantagem dos métodos estáticos, é que eles podem ser invocados diretamente de classes abstratas, que não podem ser instanciadas justamente por serem abstratas.

A **Figura 2** mostra um grupo de classes definido pela classe abstrata TExportacao e induz o pensamento de que as outras partes do projeto não dependem das subclasses deste grupo. Isto significa que o restante do projeto não usa as subclasses diretamente e instruções como TExportacaoCaixaEconomica.Create não estão espalhadas pelo projeto. Então

como partes diferentes do projeto podem usar as subclasses de TExportacao se as diferentes partes do projeto não podem instanciar estas classes invocando seus construtores?

Para conseguir criar e usar as subclasses de TExportacao, sem espalhar a dependência destas classes por todo o projeto, deve-se definir um ponto único para fabricação destas classes. Este "ponto único" na verdade é um método estático declarado na superclasse que deve ser invocado sempre que uma nova instância for necessária. Observe na **Figura 3**, que o projeto consegue exportar os dados sem conhecer qual classe concreta está sendo usada para gerar esta exportação.

Ao invés de espalhar instruções if..else pele projeto para decidir qual classe de exportação deve ser usada, um método GetExportador foi declaro na classe TExportacao para decidir qual classe de exportação usar. Este método recebe uma string como parâmetro e retorna a instância correta de acordo com o valor do parâmetro. Repare na **Figura 3** que o método de fábrica GetExportador retorna um objeto do tipo TExportacao, para fazer uso do polimorfismo garantindo que as partes do projeto consigam usar as subclasses de TExportacao sem precisar conhecê-las.

Factory Method é considerado um padrão simples, pois ele exige apenas a declaração de um método enquanto outros padrões exigem que novas classes sejam criadas. Este método torna-se o único responsável e a única forma disponível para criar instâncias de uma família de classes. Devido a esta característica, os métodos de fábrica também podem ser utilizados para reutilizar instâncias existentes ao invés de criar novas. Esta vantagem é especialmente útil para melhorar o desempenho da aplicação. Em muitas ocasiões uma aplicação não precisa de mais de uma instância de determinadas classe, como, por exemplo, as classes de conexões.

Os métodos de fábricas possibilitam criar um design mais intuitivo, pois os métodos de fábricas podem ter nomes que descrevem o que exatamente eles fabricam, além facilitar o uso do polimorfismo já que outras partes do projeto usaram as subclasses de uma família sem saber qual classe está sendo usada. Por outro lado, este

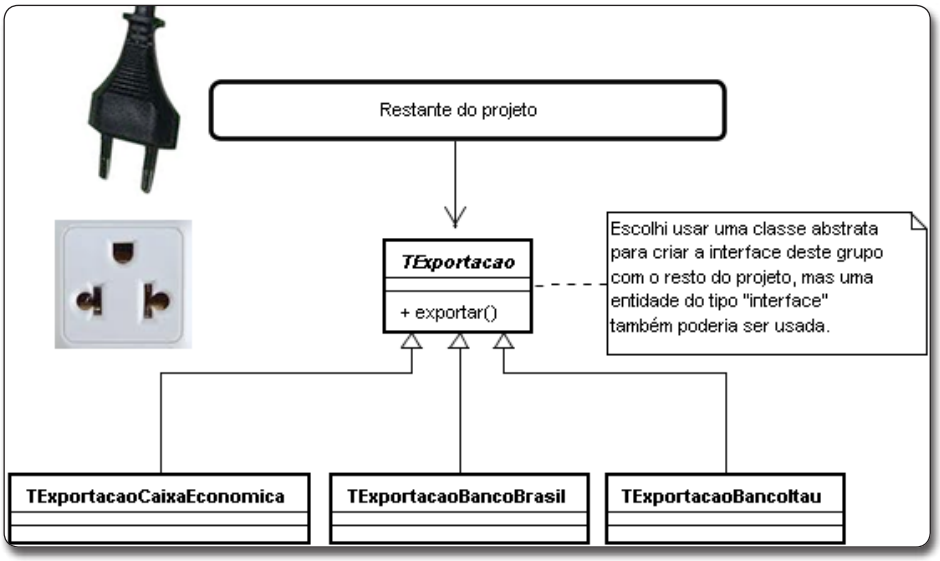


Figura 2. Criando uma interface para o grupo de classes

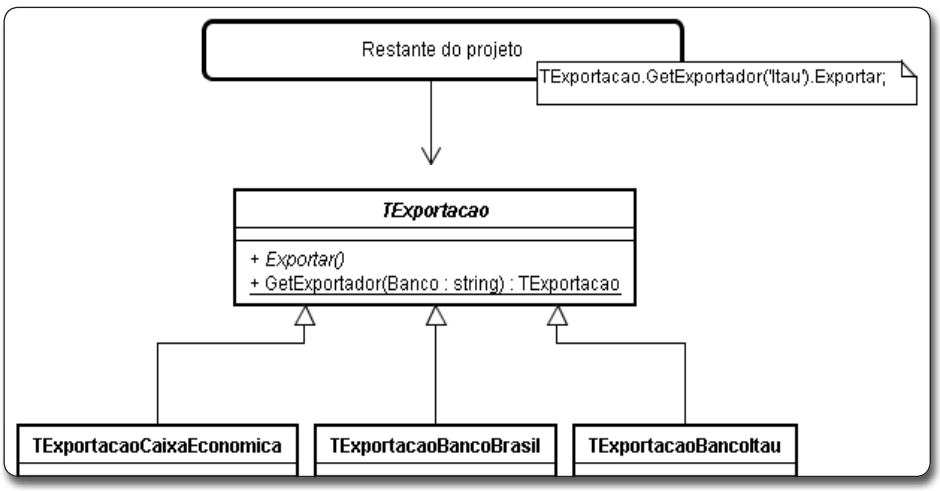


Figura 3. Declarando um método de fábrica

Nota

Um método estático não pode acessar métodos e propriedades não estáticas, ou seja, o procedimento MetodoEstatico da classe TExemplo na **Listagem 2**, não pode acessar o procedimento MetodoComum. O inverso é verdadeiro: o procedimento MetodoComum pode acessar o procedimento MetodoEstatico.

Nota

Na **Figura 3**, o método GetExportador da classe TExportacao está sublinhado para indicar que ele é um método estático.

Listagem 2. Diferenças entre método comum e método estático

```
01  type
02   TExemplo = class
03   public
04     class procedure MetodoEstatico;
05     procedure MetodoComum;
06   end;
07  implementation
08  procedure TExemplo.MetodoComum;
09  begin
10    //Código do método
11  end;
12  class procedure TExemplo.MetodoEstatico;
13  begin
14    //Código do método
15  end;

16  //===== Em outra parte do projeto =====\\
17  var Objeto: TExemplo;
18  TExemplo.MetodoEstatico;
19  Objeto := TExemplo.Create;
20  Objeto.MetodoComum;
```

padrão de projeto encoraja a proliferação da herança, o que nem sempre é bom.

Abstract Factory

O padrão Abstract Factory cria famílias inteiras de objetos. Não faça confusão com o padrão Factory Method, que cria elementos de uma família. Esta abordagem é especialmente útil para projetos que precisam ser compatíveis com plataformas diferentes. A "plataforma", neste contexto, não diz respeito somente ao sistema operacional, pode ser também o banco de dados entre outras coisas. É normal, que uma aplicação precise interagir com mais de um banco de dados diferente, neste caso, o banco de dados é a plataforma e o projeto deve ser compatível com as peculiaridades de cada um.

Cada plataforma tem uma família de classes responsável por suas funções. É como aparece na **Figura 4**, onde cada função da plataforma é encapsulada por um grupo de classes. Juntos todos os grupos formam a família de classes responsável pelas funções de determinada plataforma. O padrão Abstract Factory, define que uma classe abstrata deve conhecer e saber como criar as fábricas concretas para cada família de classes específica de cada plataforma.

Acompanhe na **Figura 5**. Uma fábrica abstrata define a interface para as fábricas concretas e cada fábrica concreta corresponde a uma plataforma. A instância de uma fábrica concreta pode ser adquirida invocando o método CriarFabricaConcreta, da classe TDatabaseAbstractFactory.

Na prática, qualquer parte do projeto que precise interagir com o banco de dados, pede para a fábrica abstrata TDatabaseAbstractFactory uma instância da fábrica concreta relacionada ao banco de dados atualmente em uso pela aplicação. Cada fábrica concreta implementa todos os métodos abstratos da fábrica abstrata. Por exemplo, a fábrica TFirebirdFactory irá retornar instâncias das classes que sabem como interagir com a plataforma de banco de dados Firebird.

Em resumo, o padrão Abstract Factory é composto por um grupo de fábricas específico para cada família de classes existentes, por isto é dito que este padrão é capaz de criar famílias inteiras de objetos. Cada fábrica é composta por um ou mais métodos de fábrica (Abstract Method), que cria o objeto responsável por encapsular as funções de cada plataforma.

No exemplo exposto, a classe TDatabaseAbstractFactory é a fábrica abstrata em si. Ela provê a interface – o contrato, como foi explicado no tópico "Programe Para Interfaces" – para o grupo de classes que sabem como fabricar os objetos específicos de cada plataforma. Todos os métodos da fábrica abstrata também são abstratos e representam uma função disponível na plataforma. Por exemplo, o método CriarGerenciadorDeConexao, deve criar e retornar o objeto que encapsula as funções relacionadas à conexão de uma plataforma de banco de dados. O único método não abstrato na classe TDatabaseAbstractFactory é CriarFabricaConcreta que assim como os demais, é um método de fábrica (padrão Factory Method). Este método cria a fábrica concreta para uma determinada plataforma.

As classes TFirebirdFactory e TOracleFactory são as verdadeiras fábricas, criadas pelo método CriarFabricaConcreta. Cada uma destas fábricas conhece quais são os grupos de classe específicos para cada plataforma, mais do que apenas conhecer, elas também sabem como criar os elementos destes grupos. Ou seja, a fábrica abstrata disponibiliza um Factory Method capaz

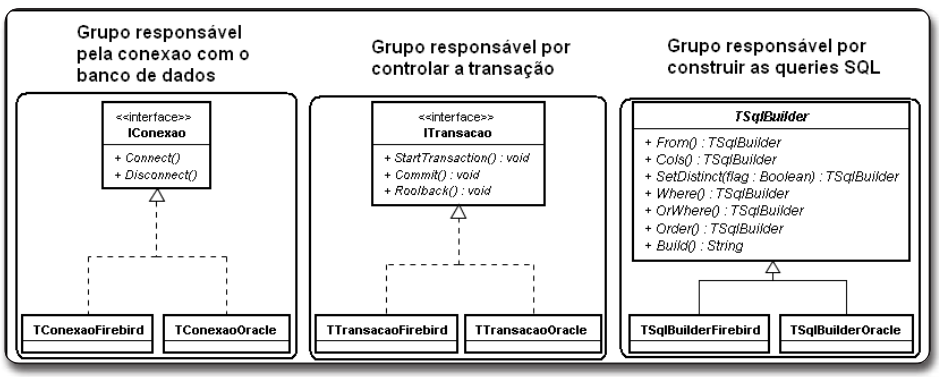


Figura 4. Definindo o grupo de classes de cada plataforma

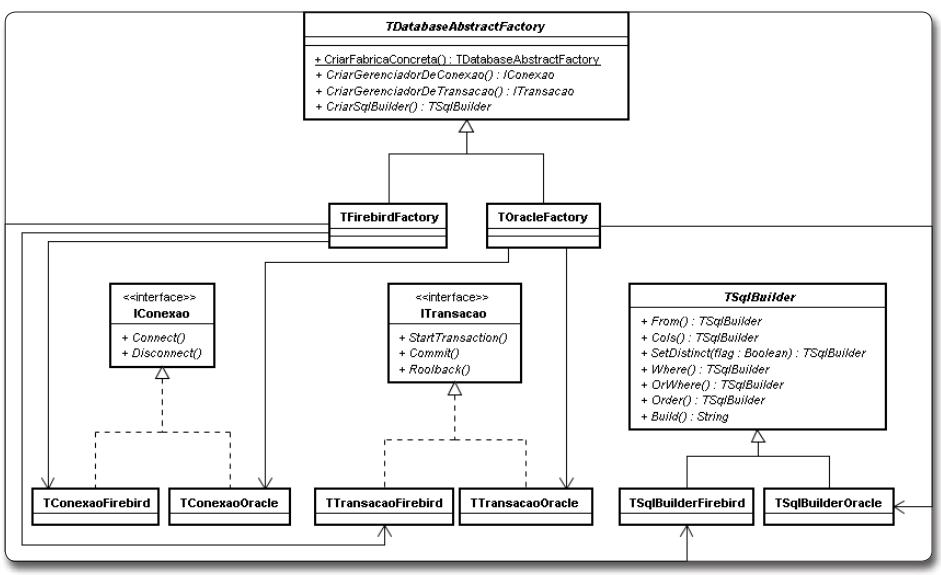


Figura 5. Implementação do padrão Abstract Factory

Nota

Nota: Você notou na **Figura 5** que dentro do padrão Abstract Factory, existe a implementação do padrão Builder? Se não notou, leia o meu artigo da edição anterior e observe com atenção o grupo de classes definido pela classe abstrata TSqlBuilder.

de criar uma fábrica correspondente a uma plataforma. Por sua vez, as fábricas concretas conhecem e criam os objetos que encapsulam as funções específicas de cada plataforma. No final de tudo, o restante do projeto dependerá apenas da classe TDatabaseAbstractFactory (a fábrica abstrata) e das interfaces (contratos) IConexao, ITransacao e TSqlBuilder.

Prototype

O padrão Prototype descreve uma forma de usar a clonagem de objetos para obter novas instâncias sem precisar invocar o construtor. Esta técnica para criação de objetos é útil principalmente quando é impossível usar o construtor de uma classe, ou quando se deseja facilitar a criação de um objeto visto que a clonagem é mais fácil e rápida do que criar e configurar todas as propriedades da nova instância. Na prática o padrão Prototype cria uma única instância de uma classe e todas as demais instâncias serão clonadas da primeira que foi criada.

A **Figura 6** mostra a estrutura deste padrão. A classe TPrototypeConcreto1 e TPrototypeConcreto2 são as classes que implementam o padrão Prototype e, portanto elas permitem que novas instâncias sejam criadas sem a necessidade de invocar o construtor: ao invés disto, pode-se usar o método Clone. A interface IPrototype é o "contrato" para oficializar a implementação do padrão, sendo assim, basta que uma classe implemente esta interface para ser considerada uma classe protótipo. Por fim, a classe TCliente representa qualquer classe interessada nas classes protótipo. Como aparece na observação ligada em TCliente, para clonar instâncias basta invocar o método Clone.

Assim como o padrão Factory Method, a implementação do Prototype define apenas um método, enquanto a maioria dos padrões precisa de uma ou mais classes. O método Clone pode ser simples ou complexo, depende da opção de quem o escreve, pois é possível usar algoritmos complexos para literalmente clonar o objeto ou apenas criar uma nova instância e configurar para ser igual a instância existente. Como o Object Pascal não oferece um método

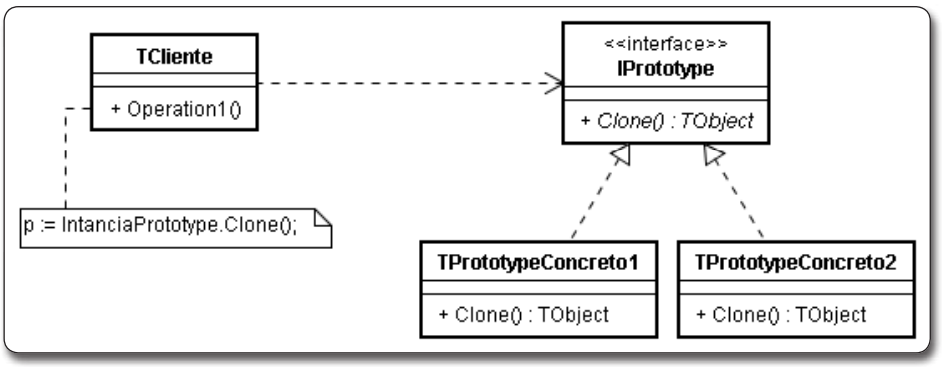


Figura 6. Estrutura do padrão Prototype

de clonagem nativo é melhor escolher a segunda opção. Veja na **Listagem 3**, como pode ser simples clonar, ou pelos menos simular a clonagem de uma instância.

A classe TPessoa (**Listagem 3**) é uma classe protótipo, pois ela implementa a classe IPrototype (linha 2) que por sua vez, obriga a implementação do método Clone. Este método, como pode ser visto entre as linhas 12 e 19, não cria um clone de verdade, ao invés disto uma nova instância é criada e os valores dos atributos da instância antiga são copiados para a instância nova. Por fim a instância nova é retornada como se fosse um clone. De fato, nenhum objeto foi clonado, mas isto não faz diferença, pois nenhuma outra classe além da própria TPessoa, saberá disto.

O padrão Prototype é um padrão criacional que, através de um método, permite criar instâncias a partir de outras instâncias. Ele não diz como deve ser implementado o método que permite criá-las, ele apenas sugere que este método receba o nome de Clone, pois afinal de contas, quando invocado ele retorna uma nova instância igual àquela cujo método clone foi invocado. Em outras palavras, não importa se um objeto realmente está sendo clonado. O que importa é que uma instância igual à instância protótipo seja retornada para a classe cliente.

Além de permitir criar novos objetos sem precisar invocar o construtor da classe, este padrão também dá ao projetista a capacidade de reciclar objetos para aumentar o desempenho da aplicação. Para isto, uma das possibilidades é guardar a primeira instância criada para ser usada para a clonagem posteriormente. Esta abordagem nem sempre pode ser aplicada, mas é uma possibilidade e normalmente precisa de um Factory Method.

Por falar em reciclagem de instâncias, existe um padrão especialista neste assunto: Padrão Singleton. Veja a seguir.

Padrão Singleton

Talvez o mais simples de todos os padrões GOF criacionais. Sua implementação depende apenas de dois fatores: proteger o construtor e criar um método estático. Não precisa nem criar interfaces ou classes abstratas. O padrão Singleton garante que somente uma única instância da classe irá existir durante a execução do sistema, bem como define um ponto de acesso global para compartilhar a única instância existente. Esta técnica pode ser comparada com a "antiga" variável global, muito comum na programação estruturada, pois assim como a variável global o padrão Singleton também compartilha uma informação com todo o projeto. Apesar da semelhança, quem já experimentou o paradigma da programação estruturada sabe o quanto uma variável global pode atrapalhar (algumas vezes também ajuda).

Uma das vantagens conseguidas com o padrão Singleton é a reciclagem de instâncias, pois uma classe Singleton determina quantas instâncias dela podem existir – normalmente apenas uma. Quando o limite de instâncias é alcançado, a classe Singleton reutiliza as que já existem. É uma grande vantagem, mas existem outras, como por exemplo, o uso de uma técnica chamada "inicialização preguiçosa". Ela tem este nome, pois as instruções de inicialização de um objeto são executadas

somente quando o objeto é necessário. O tempo de inicialização das classes que dependem de uma classe Singleton não é prejudicado, pois as classes Singleton não serão inicializadas ao mesmo tempo em que elas – a não ser que as Singleton sejam usadas durante a inicialização por algum motivo.

Classes Singleton protegem seu construtor, isto significa que nenhuma outra classe além dela própria pode invocar o construtor, o que por si só já é uma vantagem já que espalhar o uso do construtor pelo projeto pode ser prejudicial (lembre-se do princípio programe para interfaces). Quando uma classe protege o construtor, ela precisa disponibilizar alguma forma de criar instâncias de si própria, ou ela nunca poderá ser criada. Para isto um método estático – normalmente chamado de Get Instance – é declarado. Este método tem somente um objetivo: controlar a quantidade de instâncias que podem ser criadas. Este método é o ponto global de acesso à instância da classe Singleton, pois o único meio de obter a única instância existente de uma classe Singleton é invocando-o. A **Figura 7** mostra o diagrama de uma classe Singleton.

Listagem 3. Implementação do método de Clone

```
01  type
02   TPessoa = class(TInterfacedObject, IPrototype)
03   ['{BDEB1C28-D2DC-40C7-B536-1CD5BD116161}']
04   public
05    function Clone: TObject;
06    function GetNome: string;
07    function GetIdade: Byte;
08    procedure SetNome(Nome: string);
09    procedure SetIdade(Idade: Byte);
10   end;
11  implementation
12  function TPessoa.Clone: TObject;
13  var Clone: TPessoa;
14  begin
15   Clone := TPessoa.Create;
16   Clone.SetNome(Self.GetNome);
17   Clone.SetIdade(Self.GetIdade);
18   Result := Clone;
19  end;
20  // ---- Restante da classe ---- \\
```

É importante saber o que significa compartilhar a mesma instância de uma classe com todo o projeto, pois quando isto acontece, todo o comportamento da classe é compartilhado. Isto significa que um método de uma classe Singleton, é na verdade o único método existente. Por exemplo, na **Figura 7** a classe TExemploSingleton possui um método chamado Operacao1. Seja lá o que este método faz ele será o único método existente capaz de fazer. Não existirá uma coisa deste método em paralelo. Não será possível criar outra instância para ter outro objeto de acesso a este método. Isto é importante particularmente para classes Singleton que podem ser acessadas mais de uma vez ao mesmo tempo (usando TThread por exemplo).

O método Singleton – normalmente chamado de GetInstance – não é nada mais do que um simples método Get. Veja na **Listagem 4**, entre as linhas 13 e 18. O método estático GetInstance verifica se a única instância permitida já foi criada e então invoca o construtor se ela ainda não existe. Por fim a única instância existente é retornada. Repare que a instância será criada somente uma vez, por isto ela é única. Repare também que o método retorna sempre o mesmo objeto que foi criado quando ele foi invocado na primeira vez.

Para garantir que nenhuma outra classe poderá criar outra instância de uma classe Singleton, o construtor deve ser protegido. Isto é feito simplesmente declarando o construtor na sessão private, como pode ser visto na linha 5. Observe também entre as linhas 19 e 22 que este construtor não faz nada além de invocar o construtor da classe pai TObject. Mesmo que o construtor não faça nada, ele precisa ser declarado para poder ser protegido.

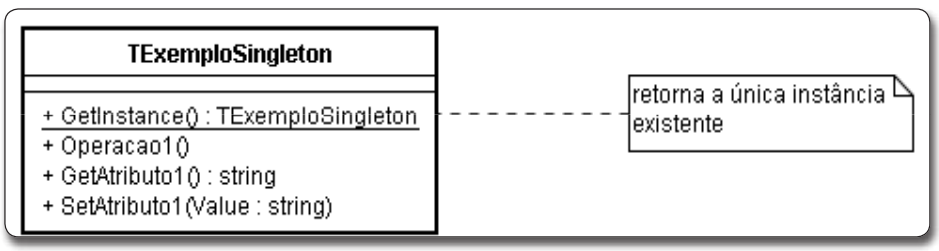


Figura 7. Exemplo de classe implementando o padrão Singleton

É importante notar a variável declarada na linha 12: ela é a única instância da classe TExemploSingleton. Ela não pode ser declarada dentro de nenhuma sessão da classe: private, protected, public ou published. Não pode ser declarada lá, pois atributos declarados dentro dessas sessões estão fora do escopo dos métodos estáticos, como é o caso do método GetInstance. Sendo assim a variável UnicaInstancia, deve ser declarada em qualquer lugar fora das sessões citadas anteriormente, mas o único lugar onde esta variável fica protegida e nenhuma outra classe poderá vê-la, é dentro da sessão implementation.

Exemplos

Para explicar como implantar um padrão vou dar continuidade ao projeto criado no artigo da edição anterior: o projeto chama-se Sistema de Pedidos. Este é um projeto utilizado em uma lanchonete e funciona da seguinte forma: um atendente da lanchonete cadastra todos os pedidos que posteriormente serão exportados para outro software da lanchonete. A exportação pode ser feita em dois formatos: XML ou TXT (para construir esta exportação foi implementado o padrão Builder).

Apesar da simplicidade do projeto, ele desrespeita muitos princípios da orientação a objetos – principalmente o princípio programe para interfaces. Veja na **Listagem 5** como um pedido é criado. O método BotaoAdicionarClick que aparece nesta listagem corresponde ao evento OnClick do botão "Adicionar ao pedido", que pode ser visto na **Figura 8**. À primeira vista parece não ser um método problemático, mas quando se presta atenção é possível ver que praticamente todo o método é constituído por uma única instrução condicional "gigante". O que acontece é que a lanchonete onde o Sistema de Pedidos é usado vende três produtos diferentes: hambúrguer, batata frita e refrigerantes. O Combo é na verdade uma venda dos três produtos juntos.

Dentro do projeto, existe uma classe para cada produto que a lanchonete vende e cada produto herda de uma classe abstrata

chamada TProdutoAbstract (veja **Figura 9**). O método BotaoAdicionarClick (**Listagem 5**) precisa criar um destes produtos para relacionar com o pedido que será cadastrado e a única forma de criar o produto certo, é usando uma instrução condicional para determinar qual o produto escolhido pelo usuário. O problema com esta prática é que o projeto está dependendo das implementações da classe TProdutoAbstract para pode criar o produto.

Para simplificar o método vou usar o padrão Factory Method. Como você viu no tópico "Factory Method" este padrão encapsula a criação dos integrantes de uma família, assim instruções como a **Listagem 5** ficarão apenas no Factory Method ao invés de ser espalhada por todo local do projeto que precise criar produtos. A interface da família de produto é definida pela classe abstrata TProdutoAbstract, o que a torna o melhor lugar para o método de fábrica. O problema é que por ser uma classe abstrata ela não pode ser instanciada e consequentemente o método de fábrica não pode ser usado, a não ser que ele seja um método estático: então a melhor saída é criar um método de fábrica estático.

Implementando: Factory Method

Compare a **Figura 9** com a **Figura 10**. Na última figura a classe TProdutoAbstract declara um Factory Method estático chamado "Factory". Ele recebe como parâmetro o nome do produto que será fabricado: "hambúrguer", "batata" ou "refrigerante" – veja na **Listagem 7** como este método decide qual produto deve ser fabricado. É um método simples que dispensa explicações.

Na **Listagem 6** aparece o método BotaoAdicionarClick depois de atualizado para usar o método de fábrica. Primeiro entre as linhas 7 e 13, o nome dos produtos que serão criados são adicionados em uma lista. Isto é necessário caso o usuário escolha o Combo, que deverá criar três produtos. O segundo e último passo é invocar o método Factory da classe TProdutoAbstract para fabricar o produto que será vendido.

Esta simples alteração permitiu que o projeto respeitasse o princípio programe

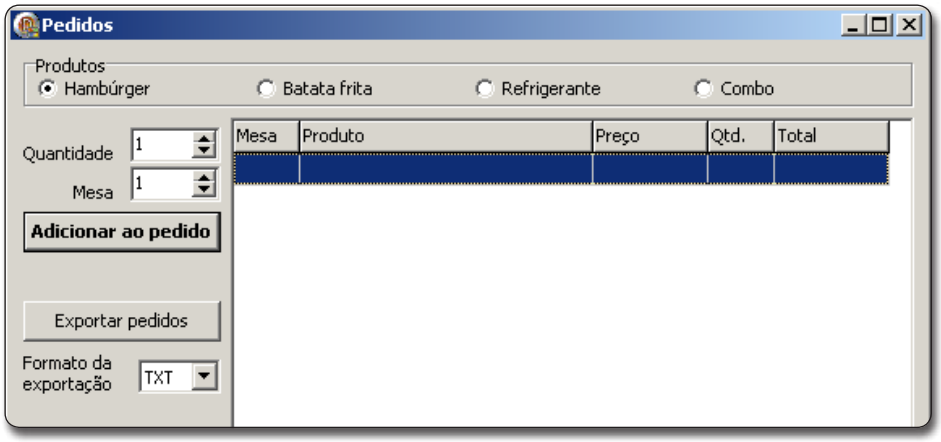


Figura 8. O formulário principal do Sistema de Pedidos

Listagem 4. Exemplo de uma classe singleton

```
01  type
02   TExemploSingleton = class
03   private
04    Attributo1: string;
05    constructor Create;
06   public
07    class function GetInstance: TExemploSingleton;
08    function GetAtrbuto1: string;
09    procedure SetAtributo1(Value: string);
10   end;
11  implementation
12  var UnicaInstancia: TExemploSingleton;
13  class function TExemploSingleton.GetInstance: TExemploSingleton;
14  begin
15   if not Assigned(UnicaInstancia) then
16    UnicaInstancia := TExemploSingleton.Create;
17   Result := UnicaInstancia;
18  end;
19  constructor TExemploSingleton.Create;
20  begin
21   inherited Create;
22  end;
```

Listagem 5. Criando um pedido

```
01  procedure TForm1.BotaoAdicionarClick(Sender: TObject);
02  Begin
03   case RadioGroupProdutos.ItemIndex of
04    0: CadastrarPedido(EditMesa.Value, THamburger.Create, EditQuantidade.Value);
05    1: CadastrarPedido(EditMesa.Value, TBatataFrita.Create, EditQuantidade.Value);
06    2: CadastrarPedido(EditMesa.Value, TRefrigerante.Create, EditQuantidade.Value);
07    3: // COMBO
08    Begin
09     CadastrarPedido(EditMesa.Value, THamburger.Create, EditQuantidade.Value);
10     CadastrarPedido(EditMesa.Value, TBatataFrita.Create, EditQuantidade.Value);
11     CadastrarPedido(EditMesa.Value, TRefrigerante.Create, EditQuantidade.Value);
12    End
13    Else
14     ShowMessage('Escolha um produto!');
15   end;
16  end;
17  procedure TForm1.CadastrarPedido(Mesa: Integer; Produto: TProdutoAbstract; Qtd: Integer);
18  var Pedido: TPedido;
19  begin
20   Pedido := TPedido.Create(Mesa, Produto, Qtd);
21   ListaDePedidos.Add(Pedido);
22   AtualizarGrid;
23  end;
```

Listagem 6. Usando um Factory Method

```
01  procedure TForm1.BotaoAdicionarClick(Sender: TObject);
02  var
03    Produtos: TStringList;
04    NomeProduto: string;
05    Produto: TProdutoAbstract;
06  begin
07    Produtos := TStringList.Create;
08    if RadioGroupProdutos.ItemIndex in [0, 3] then
09      Produtos.Add('hamburger'); //Se for Hambúrger ou Combo
10    if RadioGroupProdutos.ItemIndex in [1, 3] then
11      Produtos.Add('batata'); //Se for batata frita ou Combo
12    if RadioGroupProdutos.ItemIndex in [2, 3] then
13      Produtos.Add('refrigerante'); //Se for refrigerante ou Combo
14    For NomeProduto in Produtos do
15    Begin
16      Produto := TProdutoAbstract.Factory(NomeProduto);
17      CadastrarPedido(EditMesa.Value, Produto, EditQuantidade.Value);
18    end;
19    AtualizarGrid;
20  end;
```

Listagem 7. O Factory Method

```
01  class function TProdutoAbstract.Factory(Nome: string): TProdutoAbstract;
02  begin
03    Nome := LowerCase(Nome);
04    if nome = 'hamburger' then
05      Result := THamburger.Create
06    else if nome = 'batata' then
07      Result := TBatataFrita.Create
08    else if nome = 'refrigerante' then
09      Result := TRefrigerante.Create
10    else
11      raise Exception.Create('Tipo de produto desconhecido');
12  end;
```

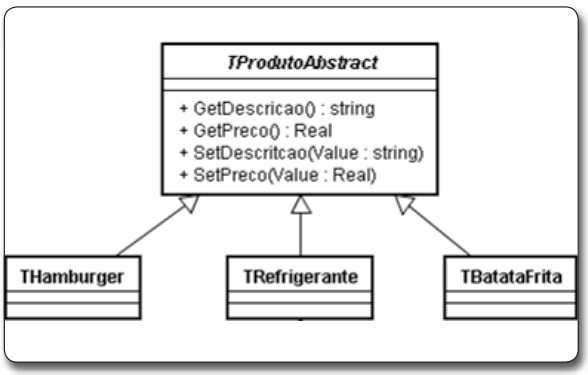


Figura 9. As classes de produto

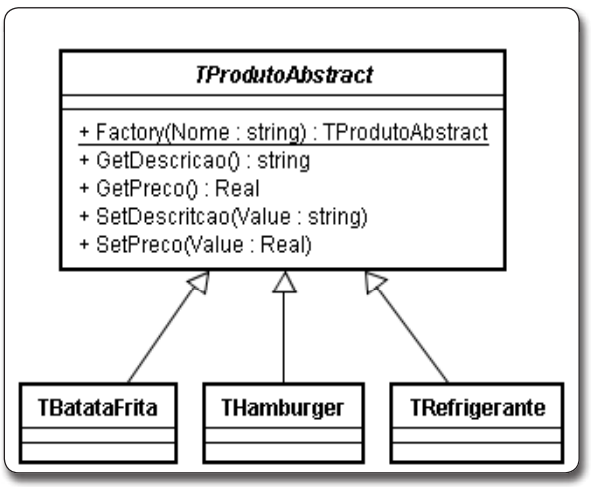


Figura 10. As classes de produto

para interface já que somente o método de fábrica conhece as implementações da classe TProdutoAbstract. Todo o restante do projeto conhece apenas a interface dos produtos, e usa as implementações sem conhecê-las. Isto é o que aconteceu no método BotaoAdicionarClick. Pois agora ele usa as classes de produtos sem saber quais são estas classes. Para ele todos os objetos de produto são representados por uma interface única chamada TProduto-Abstract.

Conclusão

Às vezes os padrões criacionais são concorrentes, em outras ocasiões eles podem ser complementares. O padrão Prototype poderia ser usado dentro do Factory Method para fabricar produtos no exemplo do Sistema de Pedidos. Além disto, Abstract Factory, Builder e Prototype podem usar Singleton na sua implementação. Também é muito comum implementar o padrão Abstract Factory com Factory Method. Sendo complementares ou concorrentes, todos encapsulam a forma como os objetos do projeto são criados diminuindo o uso descontrolado de instruções como TMinhaClasse.Create. Cada um deles possui suas próprias características. Cabe ao engenheiro "pesar" as consequências que a implementação de um padrão traz para o projeto.

Rafael Stavarengo
faelsta@gmail.com
http://stavarengo.com
Programador de sistemas da Cheina Informática com 9 anos de experiência, integrante da equipe editorial da revista Clube Delphi. Domínio em Java, PHP e UML. Sólido conhecimento em Design Patterns e metodologias ágeis. Graduado em Análise e Desenvolvimento de Sistemas pela UNIPAR.

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Acesso a dados no Delphi Prism

Rodrigo Araujo

Resumo DevMan

De que trata o artigo:

Este artigo apresentará o uso do ADO.NET Entity Framework como uma ferramenta de mapeamento objeto/relacional para realizar uma modelagem e consultas a fontes de dados, que serão visualizadas e representadas utilizando Delphi Prism em formulários Web.

Para que serve:

Serve para abstrair determinados detalhes do Banco de Dados focando na persistência de informações e possibilitando recursos que aumentem a produtividade. O uso de tecnologias para armazenamento e mapeamento do estilo objeto/relacional possibilita amenizar as diferenças entre a orientação a objetos e o modelo relacional, tornando bastante transparente e simplificada esta relação, possibilitando a quem desenvolve utilizar maneiras mais dinâmicas do esquema de um banco de dados e suas regras de negócio.

Em que situação o tema é útil:

Na utilização de um Banco de Dados relacional que possibilite tirar proveito do Entity Framework para tornar mais simples o uso das informações contidas neste banco. Dessa forma, amenizando as desigualdades entre um modelo relacional e orientado a objetos, sem que o desenvolvedor precise conhecer profundamente o modelo relacional, ampliando a produtividade.

ADO.NET Entity Framework e Delphi Prism:

O artigo demonstrará como utilizar as funcionalidades presentes no ADO.NET Entity Framework para a modelagem de um exemplo simples de Banco de Dados, suas entidades, classes, o mapeamento relacional e de objetos e o uso deste EF com o Delphi Prism. Para isto será criado um projeto do tipo Web Site, a criação do Entity Framework e adicionados Web Forms para a realização de testes com consultas e inserção de informações no banco.

Nos dias atuais, o desenvolvimento de sistemas é uma prática comumente utilizada e o que indica um bom desenvolvimento é um ótimo planejamento (análise), a competência e o prazo (tempo de desenvolvimento), entre outros fatores. A utilização de boas práticas de desenvolvimento aliadas a ferramentas que possibilitam estas práticas permitirá ganhos de produtividade e consequentemente redução do tempo de desenvolvimento, podendo assim a equipe de projeto ficar atenta às regras de análise e testes para que o sistema a ser desenvolvido seja de boa qualidade.

Ao utilizar um ambiente computacional, uma das necessidades mais básicas refere-se à persistência. Ela é a garantia da permanência das informações de uma aplicação em determinado estado, mesmo após seu encerramento. A prática mais comum para a obtenção de tal característica se faz mediante a utilização de um banco de dados relacional.

Banco de dados relacional denota um conceito, sendo este definido no SGBDR (Sistema Gerenciador de Banco de Dados Relacional), o qual possui papel muito importante na manutenção e consulta de grande volume de dados, trazendo entre outros benefícios a recuperação de falhas, velocidade nas pesquisas, acesso concorrente, integridade, independência e segurança dos dados.

A principal dificuldade gira em torno de um problema inerente à disparidade do paradigma objeto/relacional, denominada impedância de dados, que implica em um considerável esforço para transpor as diferenças existentes entre a tecnologia relacional e a orientada a objetos.

O modelo relacional é baseado em princípios matemáticos, tendo suas virtudes relacionadas à capacidade de armazenar grandes volumes de dados, provendo técnicas e conceitos adequados à garantia da integridade dos mesmos, inserindo na abordagem declarativa da linguagem SQL (Structured Query Language) uma maneira eficaz de codificar o que será feito.

Do outro lado, o modelo Orientado a Objetos está fundamentalmente atrelado à engenharia de software, evidenciando a temática imperativa da orientação a objetos, a qual faz uso de conceitos como herança, polimorfismo e encapsulamento, provando assim sua concepção no intuito de indicar como será feito.

O processo de modelagem de um sistema Orientado a Objetos e seus principais conceitos diferem na abordagem referente à tecnologia relacional, existindo muitos desafios quanto à sua utilização, incidindo no empenho em criar um esquema livre de redundâncias relacionadas ao banco de dados. Além disso, podem existir problemas relacionados a desempenho e uma série de

outras incompatibilidades, que impõem dificuldades tais como: Granularidade, Subtipos, Identidade, Associações e Navegação.

Partindo deste princípio, com o Visual Studio 2008 surgiu o Linq To Sql, uma forma de utilizar o padrão Linq para realizar consultas SQL em bancos de dados. Este foi um grande avanço que permitiu deixar de escrever instruções SQL engessadas no código e passou a utilizar queries fortemente tipadas.

Após o Linq To Sql surgiu o Entity Framework como uma ferramenta ORM. Com o Entity Framework 1.0 ocorreram mudanças na forma como os desenvolvedores e os projetos passaram a trabalhar com o banco de dados, possibilitando que a atenção fique mais voltada às regras e lógicas do negócio.

O ADO.NET Entity Framework 4 permite realizar a modelagem e o acesso as informações de um Banco de Dados deixando o desenvolvedor transparente às regras de negócio podendo trabalhar com um modelo conceitual do banco de dados. Uma das vantagens é que o mapeamento dos objetos do banco e o próprio banco de dados podem ser alterados sem prejuízo para o desenvolvimento do sistema, a camada de aplicação torna-se independente da camada de acesso a dados.

Nota do DevMan

Realizando um comparativo do Entity Framework com o Delphi tradicional, pode-se relembrar que o Delphi já teve frameworks nativos para persistência e mapeamento objeto / relacional. O Bold (Delphi 7) era usado em aplicações Win32. O ECO (Enterprise Core Objects) surgiu no Delphi 8, mas seu uso foi descontinuado na ferramenta. O Bold foi originalmente desenvolvido por uma empresa chamada BoldSoft. Em 1997 foi lançada a primeira versão do Bold para Delphi, na época, Delphi 2. Com o advento do .NET, o ECO foi lançado e teve como base o Bold, podendo ser definido como uma evolução do Bold para o mundo .NET. O ECO fez parte de um conjunto de ferramentas destinadas à Arquitetura Dirigida ao Modelo, ou simplesmente MDA (Model Driven Architecture). A MDA possui um objetivo muito claro que é tornar independente a regra de negócio de uma aplicação.

Nota

O ADO.NET Entity Framework 4 é a segunda versão do Entity Framework, adotando este nome por ser parte integrante do .NET 4.0, e, segundo a Microsoft este pulo de versões é para realizar uma sincronia de seus produtos.

Capaz de abstrair o esquema de um banco de dados relacional e apresentá-lo como um modelo conceitual, mapeando tabelas para entidades (objetos que representam dados), e preenchendo as lacunas existentes entre o modelo relacional e orientado a objetos, o ADO.NET Entity Framework é uma ferramenta ORM robusta, e suas capacidades de modelagem, acesso e manipulação de dados diminuem o ruído inerente às diferenças de impedância.

Todo o acesso a dados e armazenamento é feito através de um modelo de dados conceitual, que reflete as tabelas do banco na forma de entidades, livrando o desenvolvedor de se concentrar no seu mapeamento manual e ainda da escrita direta de consultas SQL. Essas tarefas eram tediosas e exigiam conhecimento avançado de banco de dados.

O Entity Framework é conhecido como a evolução do pacote de tecnologias ADO.NET, e sua portabilidade se estende aos mais variados bancos de dados, sendo a implementação de referência da Microsoft para o mapeamento objeto/relacional. Outra característica interessante do EF é a capacidade de suportar tipos complexos, uma novidade da última versão, bem como Lazy Loading, que permite o carregamento de entidades relacionadas conforme utilização.

O ADO.NET Entity Framework representa um grande avanço para consulta e atualização de dados relacionais, tendo em vista que as principais formas e ferramentas de manipulação de dados focavam suas estratégias em DataSet. O Entity Framework tem como principal objetivo abstrair o modelo de dados relacional em um modelo conceitual totalmente voltado a orientação a objetos, oferecendo serviços de criação, exclusão, recuperação e modificação. Apresenta um designer RAD (Rapid Application Development) para modelar entidades de dados com suporte a tipos complexos.

Ao utilizar este ORM, a equipe de desenvolvimento é encorajada a iniciar escrevendo a lógica da aplicação, fazendo uso de um modelo de dados totalmente orientado a objetos e sua arquitetura simplificada.

Com o Delphi Prism XE, Visual Studio 2010 e .NET Framework 4.0, é possível trabalhar com o ADO.NET Entity Framework, com algumas limitações. Primeiro, é que não existe, pelo menos até o fechamento desta edição, um template para criar modelos no EF baseados na linguagem Delphi Prism, como ocorre no C#. Porém, devido à integração das linguagens para .NET, é possível usar o template em C# e consumir o modelo a partir do Prism, conforme será demonstrado.

Como exemplo para o artigo será utilizado um banco de dados em branco criado no SQL Server 2008 chamado de Biblioteca, neste banco serão adicionadas as tabelas criadas na modelagem do Visual Studio 2010 utilizando os recursos do ADO.NET EF4.

Entity Framework 4

O ADO.NET Entity Framework 4, também chamado de EF4, é um Middleware (neologismo criado para designar camadas de software que fazem a mediação entre outros softwares) OR/M com capacidades de modelagem e acesso a dados, sendo uma evolução do pacote de tecnologias ADO.NET, habilitando os desenvolvedores a trabalharem com um esquema conceitual do modelo relacional.

ORM, ou Object Relational Mapping, é a técnica utilizada para mapear os dados de objetos em dados para tabelas de bancos de dados relacionais. Isto é necessário, pois em sistemas orientados a objetos informações de herança, agregações, composições e tipos complexos não são facilmente convertidos em dados relacionais.

Com o EF4, o mapeamento entre o modelo de objetos e o esquema específico de armazenamento pode mudar sem que seja necessária a alteração do código da aplicação, contando ainda com o auxílio de uma série de validações em tempo de compilação.

Possui portabilidade para os mais variados bancos de dados, sendo um concorrente direto do NHibernate, que é a versão para .NET do framework de persistência chamado Hibernate, originalmente desenvolvido para a plataforma Java.

O objetivo de um framework de persistência é "transformar" os dados que estão na aplicação na forma de objetos, em dados relacionais, compatíveis com os bancos de dados relacionais existentes.

Para que isso seja possível, é necessária a realização de um mapeamento objeto-relacional. Por este motivo o EF também é uma ferramenta de OR/M. A principal vantagem do EF é que com ele não é necessário escrever nenhum comando SQL. Todos os comandos são gerados em tempo de execução pelo framework. Além disso, ele apresenta um bom suporte para os principais bancos de dados relacionais do mercado, o que torna fácil o desenvolvimento de aplicações que trabalham com diferentes tipos de bancos.

Destacando-se do NHibernate, o Entity Framework apresenta como diferencial as várias ferramentas visuais e recursos nativos para o Visual Studio, sendo a referência da Microsoft no contexto do mapeamento objeto/relacional. Utilizando o EF4 o desenvolvedor é encorajado a iniciar escrevendo a lógica da aplicação, fazendo uso de um modelo de objetos OO, que inclui herança, tipos complexos e relacionamentos.

O EF possui um designer RAD (Rapid Application Development) para modelar entidades de dados (objetos de domínio) com suporte a tipos complexos. Na primeira versão, as classes geradas pelo O/R Designer eram intimamente ligadas ao Framework, o que dificultava a sua utilização em aplicações N-Tier (multicamadas), onde as entidades devem ser totalmente independentes com relação à maneira que serão persistidas. Existiam ainda outros pontos críticos, como a falta de suporte a POCOs, ausência de Lazy Loading e gerenciamento de concorrência, problemas resolvidos nas versões seguintes.

Criando o Entity Data Model

Através do Visual Studio 2010 é criado um Web Site utilizando a linguagem do Delphi Prism e neste Web Site, será adicionado o Entity Framework. Com o Entity Framework é possível criar primeiramente o modelo de entidade e depois criar o banco de dados (esta é a forma que será demonstrada). Neste momento, deverá ser adicionado um novo item da seção tipo Data e selecionar o template ADO.NET Entity Data Model para C# descrito na **Figura 1**.

Na próxima tela ao criar o modelo, deverá ser escolhida a opção Empty Model, com isso um modelo vazio será criado onde deverão ser definidos quais serão as entidades que posteriormente serão criadas no banco de dados.

O Empty model tem a visão de partir do modelo, ou seja, das classes de negócio, deixando a criação do Banco de Dados a cargo do Entity Framework. A técnica de iniciar o desenvolvimento pelo modelo do Banco de Dados é uma boa prática de desenvolvimento (Model-First Development, detalhado adiante), um recurso suportado pelo ADO.NET Entity Framework. A opção Generate from database é mais aconselhada para projetos legados, onde uma nova solução

Nota do DevMan

Lazy Loading é uma técnica que diz respeito aos objetos relacionados em outro objeto. Ao invés de trazer do banco de dados todas as relações do nosso objeto principal, esta carga é feita somente no momento em que esses objetos são realmente utilizados/acessados. Por exemplo, quando carregar um objeto Pedido do banco de dados, a lista de ItensDoPedido não será carregada, mas no momento em que esta lista for acessada o EF4 irá ao banco carregá-la.

Nota do DevMan

O EF e o "Vote of No Confidence"

Segundo Roger Jennings, a partir da página 353 do livro Wrox Professional ADO.NET 3.5 with LINQ and the Entity Framework, de fevereiro de 2009, em meados de 2007 alguns MVPs (Most Valuable Professionals), que mais tarde ficariam conhecidos como "NHibernate Mafia", alegaram a membros da equipe ADO.NET uma série de deficiências no design do Entity Framework, surgindo o ADO .NET Entity Framework Vote of No Confidence (voto de não confiança, uma espécie de carta digital, a qual você poderia assinar), que continha os diversos pontos onde a ferramenta poderia ser melhorada. Recorrendo novamente ao excelente livro de Roger Jennings, observa-se na página 354 uma citação, dizendo que a Microsoft estabelecera um conselho deliberativo (visando discutir os caminhos a serem tomados) tendo como base as seguintes fontes de inspiração:

- Eric Evans, autor de Domain-Driven Design: Tackling Complexity in the Heart of Software;

- Stephen Forte, diretor de estratégia na Telerik e diretor regional da Microsoft;

- Martin Fowler, autor de Patterns of Enterprise Application Software e muitos outros livros, e criador do termo Persistence Ignorance;

- Pavel Hruby, autor de Model-Driven Design Using Business Patterns and the Visio Stencil for UML and SysML;

- Jimmy Nilsson, autor de Applying Domain-Driven Design and Patterns with Examples in C# and .NET e criador das regras para Persistence Ignorance.

Nota do DevMan

POCO

POCO quer dizer Plain Old CLR Object, e significa que nossas entidades de negócio devem ser o mais simples possível, sem nenhuma dependência de ferramentas externas, como no exemplo:

```
Cliente = public class public
  property ID: System.Int32;
  property Nome: System.String;
  property Endereco: System.String;
  property Telefone: System.String;
end;
```

Veja que a classe não herda de ninguém, não implementa nenhum método e possui apenas propriedades que definem a estrutura de um cliente. Esse é o POCO mais radical que se pode fazer. Implementações menos radicais também são consideradas POCO, como por exemplo, herdar de uma entidade base, implementar uma interface, ou algum método que realize alguma tarefa específica com o cliente. Tudo isso desde que não haja nenhuma dependência com outras camadas da aplicação e ferramentas externas. Ou seja, a ideia do POCO é que tenha um modelo de entidades totalmente independente do mundo externo.

Mas qual é o benefício disso? Por que se fala tanto em ter uma camada de domínio da aplicação, desacoplada do resto? A resposta é simples. O domínio da aplicação, onde temos o modelo de entidades, assim como a modelagem das regras de negócio, é como o motor de um carro. E nós queremos construir um carro que, quando quebrar uma roda, seja necessário trocar apenas a roda e não o motor. Se você deixar o motor do carro acoplado à roda, toda vez que tiver que trocar a roda terá que trocar o motor também.

É isso que acontece em muitas aplicações hoje em dia. Muitos projetos precisam ser refeitos inteiramente porque o domínio da aplicação está espalhado entre as camadas e depende de todo tipo de tecnologia, como banco de dados, ferramentas de persistência e até da camada de apresentação. E como as tecnologias não param de surgir, onde temos novidades praticamente a cada seis meses, é importante que comecemos a planejar nossas aplicações para sobreviver a essas mudanças, senão vamos viver recriando o motor dos nossos projetos, toda vez que surge uma roda melhor no mercado.

POCO é uma boa prática voltada para isso. Por si só ele não resolve esse problema, é preciso aliar a ele uma série de boas práticas e princípios que o DDD (Domain Driven Design) ensina. Mas ele é uma das principais técnicas para atingirmos este objetivo.

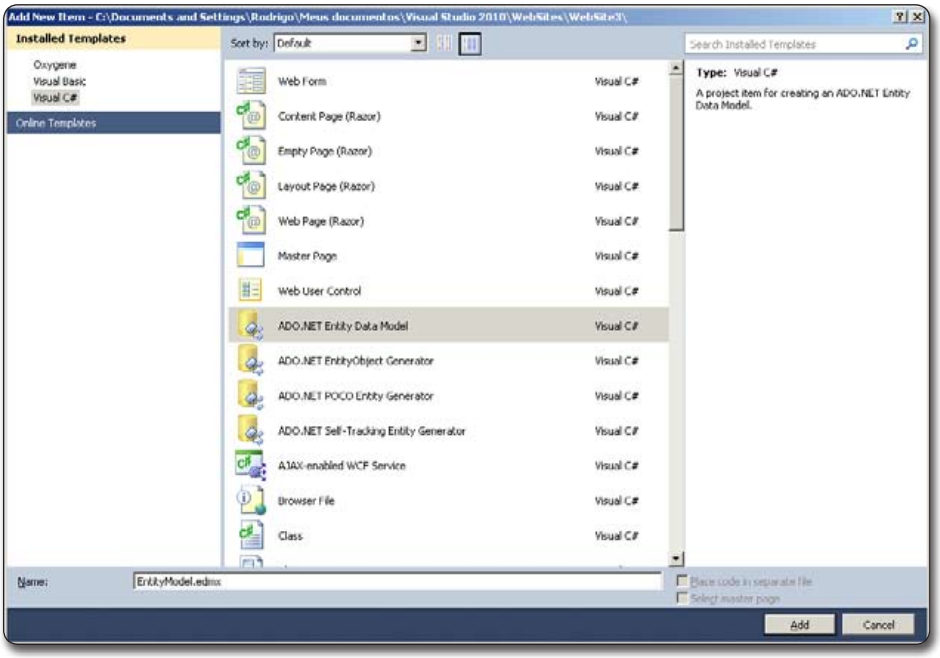


Figura 1. Criando o Entity Data Model EntityModel.edmx

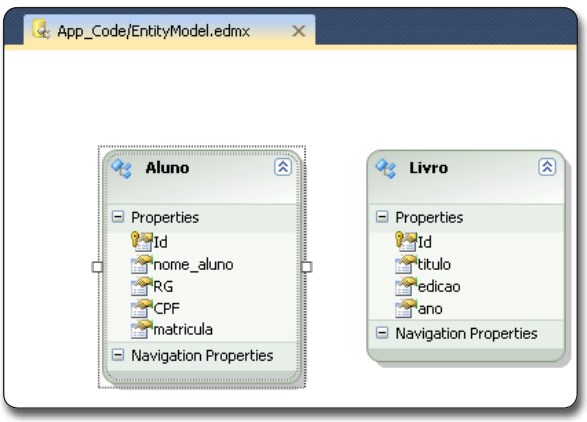


Figura 2. Modelo do Banco de Dados

Nota

Um modelo do EF é basicamente um diagrama de classes. Embora tenha sido escolhido o template C#, não será necessário digitar nenhuma linha de código na linguagem, o Visual Studio irá gerar todo o código necessário para o mapeamento. No entanto, por esta razão, não usaremos o conceito de POCO.

estará sendo feita em cima de um Banco de Dados já existente.

Talvez uma das funcionalidades mais importantes do EF é o conceito de Model-First Development. Observe que ao criar um novo modelo, existem duas opções. A criação do modelo a partir de um banco de dados, similar ao que é possível realizar com o LINQ to SQL ou os Typed Datasets. Ou a criação de um modelo vazio, e depois realizar o mapeamento Objeto/Relacional, com um banco de dados já existente.

O conceito de Model First Development, prega que primeiro deverá ser criado o modelo de entidades, e depois a criação do banco de dados baseado nesse modelo. E é esta forma que, também, está presente no Entity Framework.

Quando é criado um modelo vazio, como o que será demonstrado em exemplos no decorrer deste artigo, tem-se a possibilidade de gerar os scripts para a criação de um banco de dados baseado neste modelo.

Este exemplo funciona no modelo Web Site, onde é possível ter diferentes partes da aplicação usando diferentes linguagens, em arquivos separados na pasta App_Code. Se fosse utilizado o modelo Web Application, seria necessário criar um Assembly (Class Library) separado para hospedar o modelo.

Definindo o modelo

A **Figura 2** demonstra o modelo definindo duas entidades. A entidade Aluno e a entidade Livro, as duas contém algumas propriedades como pode ser observado.

Uma entidade nada mais é que uma tabela onde serão armazenados os dados de um banco de dados relacional. Ela é uma simples estrutura de linhas e colunas. Para uma entidade, cada linha conterá um mesmo conjunto de colunas. Em um banco de dados poderá existir uma ou centenas de entidades, sendo que o limite pode ser imposto tanto pelo desenvolvimento do aplicativo em questão ou por recursos de hardware disponíveis.

Um dos pontos principais do Entity Framework é a noção de entidade, que permite a abstração de alguma coisa reconhecida por sua capacidade independente de existência, e que pode ser identificada unicamente. É também a definição de um tipo de dados, o qual reflete algum conceito do domínio de negócio da aplicação.

Entidades são instâncias de Entity Types, e podem ser agrupadas em Entity Sets. São compostas de propriedades, que podem ser do tipo escalar (scalar), complexo (complex) ou de navegação (navigation). Existem ainda os relacionamentos (associations), que possuem suas determinadas multiplicidades.

A entidade Aluno foi criada com um Id do tipo Int32, que será um identity no DataBase (Id gerado automaticamente) e os demais campos do tipo String, além disso, a entidade possui um EntitySet, que foi criado com o mesmo nome da entidade conforme a **Figura 3**. Observe que o EntitySet é o responsável por representar a entidade em um modelo de dados conceitual.

A entidade Livro foi criada com um Id do tipo Int32, que também será um identity e os demais campos do tipo String, além disso, a entidade possui um EntitySet, que foi criado com o mesmo nome da entidade.

Criando o banco de dados

Com o modelo pronto, pode-se gerar o banco de dados. Para isso o Entity Framework disponibiliza a funcionalidade para geração do banco de dados a partir do modelo, chamada Generate Database from Model. A DDL (Data Definition Language) necessária para a criação do banco de dados será gerada automati-

camente pelo Entity Framework. Esta DDL permite definir tabelas e elementos associados. A grande parte dos bancos de dados apresentam extensões proprietárias e características próprias no DDL. Os comandos mais básicos incluem: create, alter e drop, operações básicas para criar, alterar e excluir uma tabela, uma view, um índice entre outras opções.

Observe que o Entity Framework gerou esse esquema do banco de dados, suas tabelas e os respectivos relacionamentos. A **Listagem 1** demonstra o código gerado.

Por utilizar o SQL Server 2008, propriedades do tipo identidade serão geradas como "Identity". O mapeamento de tipos também é feito automaticamente. Por exemplo, um tipo string é gerado normalmente como um varchar.

Com o Entity Framework, a preocupação não é mais com o banco de dados, o foco se fixa no modelo de negócios e nas entidades que refletem esse modelo. Resultados são retornados como objetos e os desenvolvedores não precisam se preocupar nas peculiaridades referentes aos bancos de dados. O Entity Data Model é quem descreverá o banco de dados, o modelo de classes e seu mapeamento (que fornece os metadados necessários para que o Entity Framework possa resolver os problemas de impedância). Este mapeamento expresso no EDM, possibilitará que não seja necessária a utilização de uma mesma convenção de nomes para tabelas e classes, desmembrando por completo a aplicação da fonte de dados.

Resumindo, a arquitetura planejada estará disposta em uma aplicação seguindo os conceitos de modelagem do DDD, construindo uma camada totalmente isolada e que não terá dependência com nada, a não ser o próprio .NET Framework. A camada de persistência feita com EF tem funcionalidades de criação da estrutura do banco de dados com base no mapeamento objeto/relacional definido. O exemplo utiliza o SQL Server 2008 como banco de dados, porém vale destacar que com o Entity Framework existe a possibilidade de trocar de banco de dados caso seja necessário, sem muito esforço.

Caso seja necessário no futuro, podem ser criados testes unitários para as camadas,

Nota

Além de validar o modelo, através da opção de menu, também deverá ser executada a opção Add Code Generation Item, dessa forma, validando e criando os códigos necessários para permitir o acesso através de outros locais como EntityDataSource e LinqDataSource.

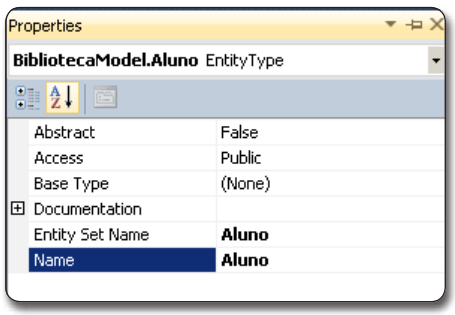


Figura 3. Propriedades da entidade Aluno

Listagem 1. Script para a criação das tabelas

```
-- ----------------------------------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, and Azure
-- ----------------------------------------------------------------------------
-- Date Created: 04/27/2011 23:15:49
-- Generated from EDMX file: C:\Rodrigo\Visual Studio 2010\WebSites\WebSite3\
App_Code\EntityModel.edmx
-- ----------------------------------------------------------------------------
 SET QUOTED_IDENTIFIER OFF;
GO
USE [Biblioteca];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- ----------------------------------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- ----------------------------------------------------------------------------



-- ----------------------------------------------------------------------------
-- Dropping existing tables
-- ----------------------------------------------------------------------------

IF OBJECT_ID(N'[dbo].[Aluno]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Aluno];
GO
IF OBJECT_ID(N'[dbo].[Livro]', 'U') IS NOT NULL
    DROP TABLE [dbo].[Livro];
GO

-- ----------------------------------------------------------------------------
-- Creating all tables
-- ----------------------------------------------------------------------------

-- Creating table 'Aluno'
CREATE TABLE [dbo].[Aluno] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [nome_aluno] nvarchar(max)  NOT NULL,
    [RG] nvarchar(max)  NOT NULL,
    [CPF] nvarchar(max)  NOT NULL,
    [matricula] varchar(15)  NULL
);
GO

-- Creating table 'Livro'
CREATE TABLE [dbo].[Livro] (
    [Id] int IDENTITY(1,1) NOT NULL,
    [titulo] nvarchar(max)  NOT NULL,
    [edicao] nvarchar(max)  NOT NULL,
    [ano] nvarchar(max)  NOT NULL
);
GO

-- ----------------------------------------------------------------------------
-- Creating all PRIMARY KEY constraints
-- ----------------------------------------------------------------------------

-- Creating primary key on [Id] in table 'Aluno'
ALTER TABLE [dbo].[Aluno]
ADD CONSTRAINT [PK_Aluno]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO

-- Creating primary key on [Id] in table 'Livro'
ALTER TABLE [dbo].[Livro]
ADD CONSTRAINT [PK_Livro]
    PRIMARY KEY CLUSTERED ([Id] ASC);
GO
```

sem nenhuma interferência direta. Logo, é possível informar que o ponto mais frágil desta arquitetura está na camada de apresentação, porém modificações e reestruturações futuras poderão ser realizadas nesta camada sem a necessidade de alteração dos demais itens do projeto.

Para finalizar, basta executar a DDL gerada pelo Entity Framework no banco a ser atualizado (ou criar um novo banco). O botão Execute SQL criará a estrutura do banco com base no código gerado pelo Entity Framework.

Definindo o acesso a dados

Para acessar os dados temos um formulário web utilizando Delphi Prism. O formulário da **Figura 4** demonstra a adição de um controle GridView e um EntityDataSource para realizar a conexão aos dados.

A **Figura 4** demonstra o uso de um EntityDataSource que representa apenas uma das várias opções de acessar e consultar dados no Entity Framework, esta foi a primeira forma desenvolvida para o Entity Framework, pois quando este iniciou, a LINQ estava apenas em fase de protótipo. Durante o desenvolvimento do EF, a LINQ ficou estável e o dialeto LINQ to Entities foi disponibilizado.

A ideia inicial utilizada por um Entity DataSource usa um Entity SQL para criar uma linguagem de consulta que fosse simples para o desenvolvedor entender, e semelhante ao SQL. Conforme o Entity Framework foi evoluindo, novas funções foram necessárias para a linguagem.

Entity SQL está na frente do LINQ to Entities em algumas situações, pois por ser baseado em strings, é mais fácil de criar consultas com condições baseadas em tempo de execução. Outro ponto é que Entity SQL é a única linguagem que pode ser utilizada para retornar dados em baixo nível, usando diretamente o Entity Client Data Provider. Existem ainda funções que não podem ser invocadas usando LINQ to Entities, mas somente com Entity SQL.

Além disso, o Entity Client é responsável pela comunicação com o banco de dados, e para que sua arquitetura seja mais simples, ele depende da infraestrutura do provider ADO.NET. Sua responsabilidade é traduzir as consultas baseadas no modelo de entidades para SQL e transformar o resultado tabular do banco de dados na forma conceitual.

A execução de uma consulta é traduzida, através do Object Services, em uma árvore de comandos, que então é repassada para o Entity Client Data Provider, processo esse conhecido como Query Transformation. Se a linguagem de consulta utilizada for o LINQ to Entities, o provider LINQ irá gerar antes uma árvore de expressões, que é analisada e transformada em uma árvore de comandos. Se a consulta tiver sido desenvolvida com Entity SQL, que se utiliza de strings, a string passada é analisada e transformada em uma árvore de comandos.

Esta camada pode ser utilizada diretamente, maximizando o desempenho das consultas. Isto se deve principalmente por evitar o processo de materialização, que é a tarefa mais lenta na execução de uma consulta, e deve ser feito apenas quando o desempenho é muito crítico. Entity SQL é a única linguagem que pode ser utilizada neste cenário, pois o LINQ to Entities manipula objetos, e nesta camada eles são desconhecidos.

De maneira semelhante, a **Figura 5** indica outro WebForm criado, mas desta vez utilizando um LinqDataSource para acessar as informações.

LINQ to Entities é um dialeto do LINQ (Language Integrated Query), e permite que o desenvolvedor escreva consultas baseadas no modelo de dados a partir da mesma linguagem utilizada para construir a lógica de negócio. Mesmo que o LINQ to Entities seja operado em objetos, ele ainda precisa ser traduzido para SQL, que finalmente é lançado no banco de dados. O objetivo da Microsoft com a criação da Language-Integrated Query (LINQ) foi aproximar o mundo dos objetos do mundo dos dados. O LINQ corresponde a uma sintaxe unificada, inicialmente incorporada às linguagens C# e Visual Basic (hoje também no Prism), para consultas em fontes de dados variadas. A sintaxe de consulta da LINQ foi inspirada na da Structured Query Language (SQL), que é uma linguagem padrão para comunicação

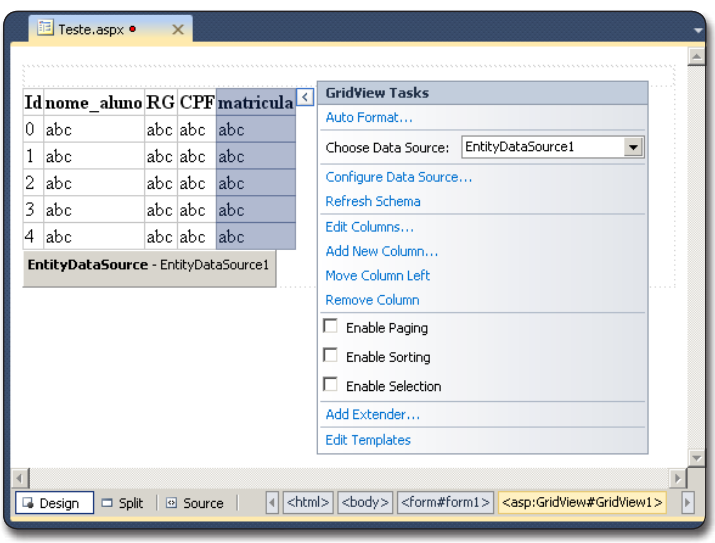


Figura 4. Tela de consulta para as informações dos alunos através de um EntityDataSource

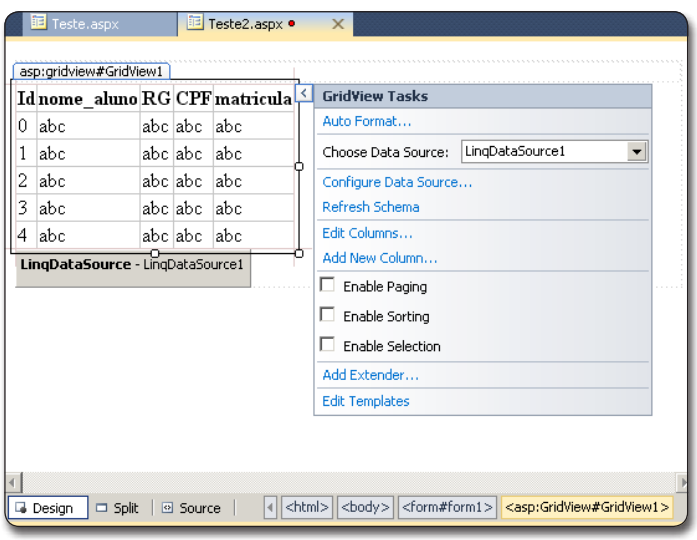


Figura 5. Tela que consulta as informações dos alunos através de um LinqDataSource

com bancos de dados relacionais. Assim como na linguagem SQL, as expressões de consulta LINQ permitem a construção de instruções variadas para extração de informações.

Tradicionalmente, as consultas a bancos de dados em aplicações eram expressas por strings sem verificação de sintaxe nem em tempo de projeto e nem em tempo de compilação, além de não haver suporte a IntelliSense. Erros de sintaxe na consulta somente eram identificados em tempo de execução. Além disto, o desenvolvedor precisava aprender linguagens de consulta específicas para cada tipo de fonte de dados, como: bancos de dados relacionais, coleções de objetos na memória, documentos XML, dentre outros tipos de fontes de dados. A ideia com a LINQ foi tornar as consultas como um recurso de primeira classe nas linguagens de programação da plataforma .NET.

Muitos dos métodos e sobrecargas do LINQ não podem ser representados em SQL. Apesar disso, são sintaticamente válidos, e uma chamada a um deles não irá causar erro em tempo de compilação. A **Listagem 2** demonstra uma consulta LINQ para ser utilizada com a entidade Livros em Delphi Prism.

As expressões de consulta LINQ permitem uma integração mais visível com a linguagem do Delphi Prism. Estas expressões são escritas com uma sintaxe declarativa que permite realizar, com um mínimo de código, diversas operações em dados, como filtragens, projeções, ordenações, agrupamentos e junções.

Da mesma forma, a **Listagem 3** permite visualizar um código em Prism para incluir um novo Livro no banco de dados. Observe que ao utilizar LINQ, não ocorre em nenhum momento um comando SQL, nem INSERT, aqui estará sendo representado POO. Assim, estará criado um objeto fortemente tipado e será realizada a requisição à variável de contexto para adicioná-lo (Add Object), chamando a seguir SaveChanges (o "ApplyUpdates" do EF).

O padrão LINQ torna simples a consulta em fontes de dados para as quais o LINQ esteja habilitado, uma vez que a sintaxe e o padrão de consultas não muda. Um LINQ Provider para uma determinada fonte de dados permite não somente a operação de consulta, mas também operações de inclusão, atualização e exclusão, além de mapeamento para tipos definidos pelo usuário. O .NET Framework suporta vários LINQ Providers tais como LINQ to Objects, LINQ to XML, LINQ to DataSet, LINQ to SQL e LINQ to Entities.

O Entity Framework possui um provider público, que torna possível suportar providers de terceiros, tais como Oracle, MySQL, PostgreSQL e Firebird. Este modelo de provider fornece uma maneira uniforme para o Entity Framework consultar recursos do banco de dados. Sempre que uma consulta LINQ to Entities ou Entity SQL ocorre através de uma instância de Object Context, esta irá passar por uma série de camadas de comunicação e, de uma maneira genérica, pode-se dizer que todas as consultas e atualizações de Object Context serão traduzidas e executadas através de uma Entity Connection, que é responsável por comunicar-se com um provider em específico. ObjectContext é a classe principal para as iterações de objetos que são entidades de um modelo. Um objeto do tipo ObjectContext encapsula uma conexão com o banco de dados na forma de uma EntityConnection. Possui ainda os metadados de mapeamento entre as entidades e o banco de dados e um objeto do tipo ObjectStateManager que mantém informações sobre alterações nas entidades, além de saber o que deverá ou não ser persistido/atualizado no banco de dados.

Nota do DevMan

Data Binding

A utilização de WebForms juntamente com o Visual Studio 2010, permite usar recursos excelentes com maior facilidade para configurar o Data Binding de um formulário. Basicamente são arrastados alguns controles e tudo é criado automaticamente.

As técnicas de Data Binding fornecem uma maneira automática para mostrar e fazer alterações nas informações de fontes de dados nos controles de um formulário. É um processo capaz de realizar a vinculação de informações, o que facilita a vida do desenvolvedor.

Caso não existisse o processo de Data Binding, seria necessário monitorar alterações em controles de tela, geralmente através de eventos, gerar novos comandos para atualizar dados, recuperar informações e assim por diante.

Com Data Binding o .NET garante que os dados exibidos nos controles de tela serão obtidos automaticamente da fonte associada. Alterações realizadas também serão automaticamente devolvidas à fonte original, processo conhecido como "two-way data binding", ou seja, de duas vias. Uma alteração na fonte reflete no controle, uma alteração no controle reflete na fonte.

Testes

Após implementar as páginas web, os testes básicos de inclusão representados por LINQ, e o uso de LinqDataSources e EntityDataSources, o WebForm Teste apresentará um comportamento semelhante ao da **Figura 6**.

Figura 6. Tela Web com Delphi Prism

Listagem 2. Definindo uma consulta LINQ

```
method Teste.Page_Load(sender: Object; e: EventArgs);
begin
  var ctx := new EntityModel.ModelContainer();
  var qry := from C in ctx.Livro
        select C;
  GridView2.DataSource := qry;
  GridView2.DataBind();
end;
```

Listagem 3. Usando POO para incluir um novo Livro

```
method Teste.Page_Load(sender: Object; e: EventArgs);
begin
  var ctx := new EntityModel.ModelContainer();
  var Liv := new EntityModel.Livro();
  Liv.titulo := "Teste de Título";
  Liv.edicao := "Ed. 3";
  Liv.ano := "2011";
  ctx.AddObject("Livro",Liv);
   ctx.SaveChanges();
end;
```

Conclusão

O ADO .NET Entity Framework representa um grande avanço da Microsoft para consulta e atualização de dados relacionais, tendo em vista que desde 2002 as ferramentas de manipulação de dados focavam suas estratégias no DataSet.

Além disso, é uma ferramenta OR/M robusta e, em comparação com outras tecnologias similares, exige uma curva de aprendizado muito pequena, pois é intuitiva e sua integração com o compilador é total. Com a utilização do LINQ to Entities, o desenvolvimento se torna mais padronizado, o que gera ganho de produtividade e facilidade de manutenção, haja vista que ele ajuda a isolar o banco de dados da aplicação, melhorando sua portabilidade e entendimento.

Assim, pode ser observado como é possível realizar uma modelagem, criação de um Banco de Dados simples e o uso de classes utilizando ADO.NET Entity Framework 4 no Visual Studio 2010 focado para ASP.NET com o Delphi Prism. As classes desenvolvidas foram criadas de maneira simples para apenas demonstrar como as funcionalidades poderiam ser observadas. A criação do Banco de Dados foi realizada a partir da modelagem, mas o banco poderia ter sido todo criado no SQL Server 2008, mas o destaque foi dado ao Visual Studio para demonstrar que também pode ser utilizado com esta funcionalidade já preparando e indicando os tipos das Entidades. Dessa forma, ao utilizar LINQ to Entities um dos objetivos é diminuir o problema da impedância existente entre Banco de Dados relacionais e as linguagens orientadas a objeto, possibilitando mapear tabelas e oferecer uma sintaxe de consulta próxima da linguagem de quem estará desenvolvendo.

Para concluir, o artigo expôs uma forma de utilização do Entity Framework em conjunto com ASP.NET voltado para a linguagem do Delphi Prism, permitindo a utilização de classes pré-estabelecidas e a demonstração de como acessar dados e realizar operações de inclusão de dados, cabendo a quem desenvolve verificar qual a forma mais apropriada para criar um sistema.

Rodrigo Araujo
rodrigo.arsi@gmail.com
Bacharel em Ciência da Computação pela UFSM – Santa Maria/RS trabalha com Análise e Desenvolvimento de Software há mais de 10 anos. Desde 2003, utiliza tecnologias .NET.

Microsoft ADO.NET Entity Framework Feature Community Technology Preview *http://www.microsoft.com/downloads/details.aspx?FamilyID=af18e652-9ea7-478b-8b41-8424b94e3f58&displayLang=en*

What's New and Cool in Entity Framework 4.0 *http://www.develop.com/entityframework4*

MSDN - Entity Framework Overview *http://msdn.microsoft.com/en-us/library/bb399567.aspx*

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

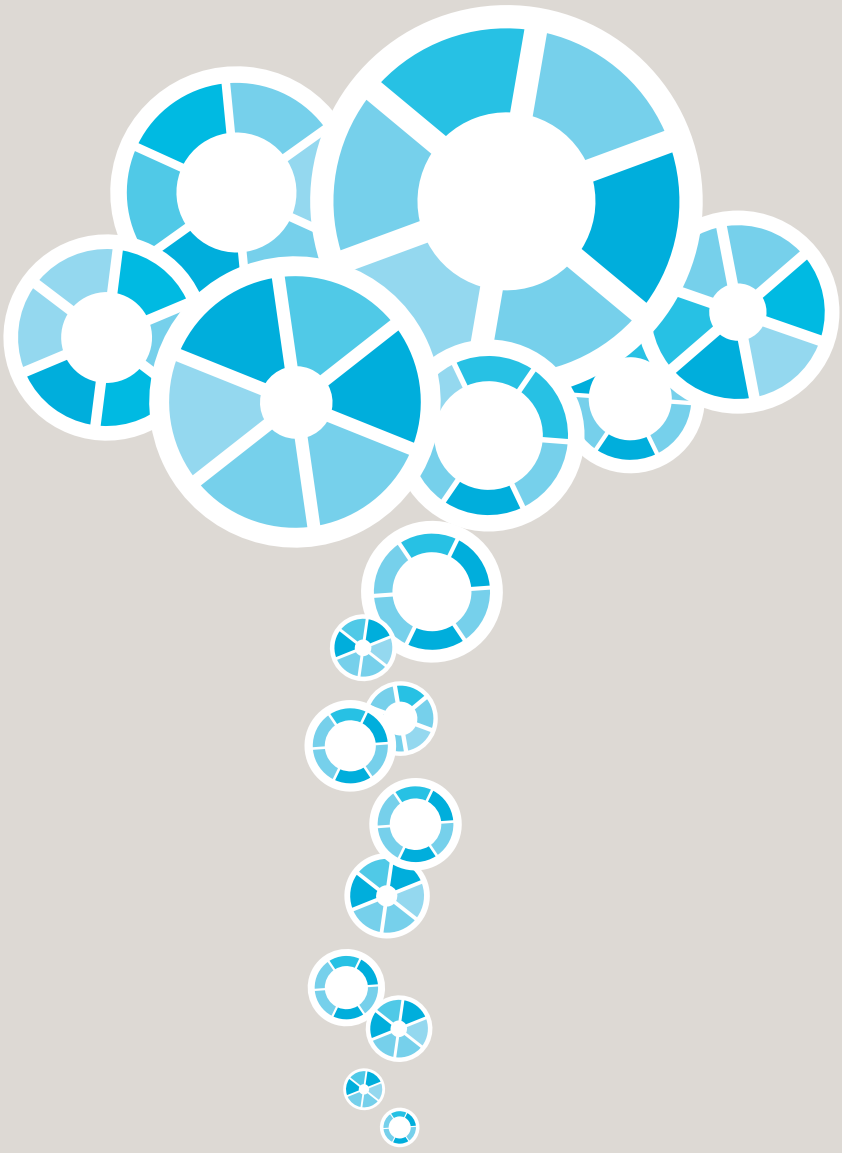
Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.

Microsoft

wunderman



Quem quer novas possibilidades pensa em Windows Azure.

Windows Azure é uma plataforma de desenvolvimento que está revolucionando a forma de se pensar no assunto. Com ela, você desenvolve e hospeda aplicativos diretamente na Nuvem. Assim, você os acessa de uma maneira rápida e fácil de qualquer lugar, através da nossa Rede de Fornecimento de Conteúdo. Possibilidade de trabalhar de qualquer lugar. Este é o Poder da Nuvem.

Teste o Windows Azure. É grátis.
Acesse Microsoft.com.br/nuvem/desenvolvedor

Para ler o código ao lado, baixe grátis nosso aplicativo para celular em http://gettag.mobi e acesse o site mobile. Para assinar esta oferta será necessário um cartão de crédito (nacional ou internacional). Ao exceder os montantes definidos* haverá cobrança dos serviços com base na taxa regular. Esta oferta está disponível até o dia 30 de setembro de 2011 (23h59 UTC).

*Mais informações: http://www.microsoft.com.br/nuvem/desenvolvedor

Windows Azure

Microsoft é marca ou marca registrada da Microsoft Corporation nos Estados Unidos e/ou em outros países.